Final CONNECT Architecture

Emil-Mircea Andriescu, Amel Bennaceur, Gordon S. Blair, Antonello Calabro, Paul Grace, Gang Huang, Valerie Issarny, Massimiliano Itria, Yun Ma, Charles Morisset, et al.

CONNECT

Emergent Connectors for

Eternal Software Intensive Networked Systems

ICT FET IP Project

Deliverable D1.4

Final CONNECT Architecture

SEVENTH FRAMEWORK PROGRAMME

http://www.connect-forever.eu

Inria · DOCOMO Euro-Labs · Ambientic · Lancaster University · Thales · Universita' dell'Aquila · technische universität dortmund · University of Oxford · Uppsala Universitet

Project Number	:	231167
Project Title	:	CONNECT – Emergent Connectors for Eternal Software Intensive Networked Systems
Deliverable Type	:	Report

Deliverable Number	:	D1.4
Title of Deliverable	:	Final CONNECT Architecture
Nature of Deliverable	:	R
Dissemination Level	:	Public
Internal Version Number	:	20
Contractual Delivery Date	:	30th November 2012
Actual Delivery Date	:	17th December 2012
Contributing WPs	:	WP1
Editor(s)	:	Paul Grace, Gordon S. Blair (LANCS)
Author(s)	:	Emil-Mircea Andriescu (Ambientic, Inria), Amel Bennaceur (Inria), Gordon S. Blair (LANCS), Antonello Calabro (CNR), Paul Grace (LANCS), Gang Huang (PKU), Valerie Issarny (Inria), Massimiliano Itria (CNR), Yun Ma(PKU), Charles Morisset(CNR), Vatsala Nundloll (LANCS) , Pierre-Guillaume Raverdy (Ambientic), Rachid Saadi (Ambientic), Roberto Speicys Cardoso (Ambientic), Daniel Sykes (Inria)
Reviewer(s)	:	Thales, Inria

Abstract

Interoperability remains a fundamental challenge when connecting heterogeneous systems which encounter and spontaneously communicate with one another in pervasive computing environments. This challenge is exasperated by the highly heterogeneous technologies employed by each of the interacting parties, i.e., in terms of hardware, operating system, middleware protocols, and application protocols. The key aim of the CONNECT *project is to drop this heterogeneity barrier and achieve universal interoperability. Here we report on the revised* CONNECT *architecture, highlighting the integration of the work carried out to integrate the* CONNECT *enablers developed by the different partners; in particular, we present the progress of this work towards a finalised concrete architecture. In the third year this architecture has been enhanced to: i) produce concrete* CONNECT*ors, ii) match networked systems based upon their goals and intent, and iii) use learning technologies to find the affordance of a system. We also report on the application of the* CONNECT *approach to streaming based systems, further considering exploitation of* CONNECT *in the mobile environment.*

Keyword List

Interoperability, middleware, middleware heterogeneity, service discovery heterogeneity, service interaction heterogeneity, application-level heterogeneity, data heterogeneity, non-functional properties, software architecture, connectors, semantics, ontologies.

Document History

Version	Type of Change	Author(s)
1	Document creation	LANCS
2	Introduction	LANCS
3	Ontology handbook chapter	INRIA
4	Chapter 2, 3, and 4 - architecture details	LANCS, CNR, PKU
5	Chapter 6 Initial evaluation	LANCS
6	Chapter 7 conclusions	LANCS
7	Revised version based on internal review	LANCS
8	Chapter 6 final evaluation	INRIA
9	Chapter on mobile streaming	Ambientic
10	Final Version	INRIA, LANCS

Document Review

Date	Version	Reviewer	Comment
03/12/2012	8	Youssouf Mhoma (THALES)	General editing errors to fix. Improvement of the evaluation with further content.
07/12/2012	8	Valerie Issarny	1) Improvement of ontology section required. 2) Include further content about mobile streaming. 3) Align evaluation with D6.3 assessment. 4) Future work is too Lancaster centric, make project wide.

Table of Contents

List of Figures

1 Introduction

Interoperability is the ability of systems developed independently of one another to exchange, understand and use each other's data. In environments where networked systems (NS) are highly heterogeneous and spontaneously interact with one another, achieving interoperability remains a challenging problem; pervasive computing, and complex systems-of-systems highlight domains where these interoperability issues are faced today.

The aim of the CONNECT project is to identify new approaches to address this interoperability challenge, and solve the problem in a fundamentally different way. That is, rather than predefine a particular global standard or middleware solution, the interoperability software is created dynamically to meet the requirements of the interoperating systems. At the heart of the CONNECT approach is the utilization of technologies not typically associated with middleware software. Learning and discovery technologies first build a picture of the structure and behaviour of the networked systems wishing to interoperate. Based upon this information a CONNECTor is dynamically synthesized that will ensure the two systems will interoperate. The CONNECTor is then monitored to ensure it is maintaining the required non-functional properties (with respect to dependability, security and trust).

1.1 The Role of Work Package WP1

The role of WP1 is to provide an overall architecture for CONNECT, and in particular define and document the common architectural principles behind the CONNECT solutions to achieving extremely long-lived (eternal) networked systems. The original three tasks of WP1 as described in the description of work [11] are as follows:

Task 1.1: CONNECT architecture. Elaborating a technology-independent and eternal architectural framework for emergent CONNECTors.

Task 1.2: Eternal system semantics. Eliciting an ontology-based characterization of the semantics of connected systems.

Task 1.3: CONNECT realization. Developing key underlying systems principles and techniques to support the development of a practical, efficient and a self-sustaining CONNECT prototype.

Hence, this work package performs a central role within CONNECT as a whole: acting as a point of integration for the specialised work from each of the work packages. Importantly, WP1 provided the system prototypes necessary to directly support the experimentation and evaluation work of the project as a whole.

1.2 Summary of Achievements in the Project and Year Four

1.2.1 Overall Project Achievements in WP1

In the first stage of the project (reported in Deliverable D1.1 [5]) we identified five key types of heterogeneity that act as a barrier to interoperability: i) *discovery protocol*, ii) *middleware protocol*, iii) *data heterogeneity*, iv) *API heterogeneity*, and v) *heterogeneous non-functional properties*. Our survey of existing research and industrial solutions [5] then showed that no approach proposed by either the middleware or the semantic interoperability communities achieved a complete solution that addressed each of these five types of heterogeneity in dynamic, pervasive computing environments.

Deliverables D1.1 [5], D1.2 [2], and D1.3 [3] presented and refined the CONNECT architecture. Here a set of key architectural principles realise the CONNECT vision:

- The CONNECT *Networked System Model* defines a runtime software artefact that accurately describes each networked system in order to inform the CONNECT process. The model instances contain a rich semantic description of each individual networked system in terms of their: role, interface syntax, behaviour and non-functional properties.

- The *Enabler Architecture* describes how enablers (software components that perform the CONNECT process of creating CONNECTors) are composed and co-operate to complete the functionality of CONNECT. The Enabler Architecture is split into two phases: i) the CONNECT*ion phase*, which performs the initial behaviour required to generate a new CONNECTor; and ii) the CONNECT*ability phase* which monitors and adapts the CONNECTor in order to maintain dependability, performance, security and trust requirements.

- The CONNECTor architecture describes how the software to connect two networked system is constructed (that is what is produced by the CONNECT process).

- The identification of how ontologies cross-cut the CONNECT architecture, and in particular how they are specified, also how they are utilised by the individual enablers

Further, WP1 concentrated on the integration of the work from across work packages to form a concrete realisation of the above architectural principles, and importantly the corresponding software to support the evaluation phase of CONNECT. Deliverable D1.3 [3] illustrated the key achievements in this area (which have been further refined in the third year):

- *From abstract to concrete* CONNECT*ors*. The Starlink toolkit [10] supported the concept of concrete *k-Coloured automaton* that could be executed to mediate between networked systems using their legacy middleware protocols. This allowed CONNECT to synthesize executable CONNECTors.

- *Implementation of an Integrated Architecture*. The Discovery and Learning enablers were enhanced to include: affordance learning, and the discovery of networked system goals; and the CONNECT architecture was enhanced to manage non-functional requirements. In particular the CONNECTors were instrumented in order to inform the monitoring of behaviour.

- *Application of the* CONNECT *approach to media streaming applications in the mobile environment*. Networked systems providing streaming services, e.g., video streaming using a streaming communication protocol (e.g., RTSP) offer different challenges for CONNECTor construction in order to address interoperability. Streaming CONNECTors demonstrated how CONNECT solutions achieve interoperable streaming by mobile devices.

1.2.2 Achievements in Year Four.

In the final part of the project, the work in WP1 concentrated on the enhancement and implementation of the architecture especially with respect to CONNECTability capabilities. Furthermore, WP1 focused on the task of evaluating the CONNECT architecture from the viewpoint of correctness of generated CONNECTors and the potential long-term impact the architecture has on the field of distributed systems.

The key achievements in the final phase are:

- CONNECT*ability*. The architecture was enhanced to support the synthesis of CONNECTors with security properties. Here, security policies (developed within WP5) capture the required secure behaviour of a generated CONNECTor. The Enabler architecture was refined to support the instrumentation of synthesized CONNECTors with these policies. This is reported in Section 3.2.4 of this deliverable and also in Deliverable D5.4 [15].

- *Alternative middleware styles*. Previously our work focused on interoperability between networked systems that only employed RPC-based middleware protocols. We investigated alternative styles, and in particular built suitable implementation to achieve interoperability between networked systems involving publish-subscribe middleware. This was a significant result as it identified the increased breadth of the CONNECT solutions. The case study in Section 7.3.1 highlights this work.

- *Evaluation*. The final architecture was evaluated using interoperability case studies; this highlights how the CONNECT solution achieves correct interoperability. We then returned to the state of the art presented in Deliverable D1.1 [5] and analyzed CONNECT against prior work to show how it exceeds any existing middleware or semantic web solution.

- *Adaptation*. Work was carried out to investigate approaches to adapt deployed CONNECTors within the CONNECT architecture. The use of Models@runtime are presented as an ideal approach towards achieving eternal CONNECTor solutions.

1.3 Review Recommendations and Challenges for Year Four

The project reviewers proposed the following recommendations for both the WP1 work package and the project in light of the review of the work carried out during the third year of the project:

"While the abstract architecture for the connector synthesis and the supporting tools have been defined and presented clearly, the deployment of these components in a real environment is less clear. Some open questions are: Where are the various enablers located at? Can there be distributed enablers and how about interoperability of heterogeneous enablers? Where and how is the monitoring performed? How to deal with ontology-based reasoning on mobile devices? The project should be more concrete about the deployment options for CONNECT *components."*

In the final version of the architecture and its implementation, the enablers exist as singleton components deployed on CONNECT hosts. This means that the enablers can be deployed across one or more hosts; and in the demonstrator presented in Deliverable D6.4 [14] we describe how this is practically realised. The only requirement is that the enablers are connected. Furthermore, only the discovery enabler need be within the same network domain as the networked systems (because it employs multicast to discover behaviour). In the fourth year we have examined approaches to manage the enabler architecture itself, e.g. with respect to load balancing and this work is presented in Section 4.3.2.

While the idea of heterogeneous enablers is an interesting problem, at present we only consider the case of one version of each of the enablers used by the CONNECT architecture.

"A recommendation of the 2nd review was to explain how the architecture deals with adaptation in case of changing quality of service. The relationship between connector synthesis, QoS monitoring and connector adaptation is still not clear and requires further attention and explanations."

Section 4 finalizes the conceptual vision of how CONNECTability is realised, and in turn how the CONNECT architecture provides the mechanisms to perform adaptation. Two papers provided in the appendix then further explore how CONNECT adaptation can be performed using adaptation of the CONNECT models that are available at runtime.

"Most of the complexity of dealing with heterogeneity in large-scale distributed environments has been dumped on the domain ontology. In order to facilitate the future take-up of CONNECT *results, the project should come up with a concise description of the role, structure, contents, management, evolution, responsibilities etc. of the domain ontology in the* CONNECT *framework. This description could be in the form of a "How-to-Guide"."*

Section 5 provides such a "How-to-Guide" and offers a clear explanation of how ontologies can be leveraged to resolve the issues of interoperability.

"The implementation of some of the enablers seems to be unnecessarily restricted. For example, higher level data transformations (such as jpg to gif) are not foreseen. There are partners in the consortium who have already infrastructure in place to elegantly deal with the generation of a larger class of mediators than considered in the project. Why isnt this taken over in the project?"

The CONNECT architecture provides the mechanisms to insert such functions directly into the concrete k-coloured automaton model using the translation logic defined in Section 2.2.2. Hence, provided these higher level data transformations are made available and are documented in the appropriate ontologies, it is then possible for them to be included.

"It should be clarified to which extent is the proposed way of monitoring in distributed environments viable. A Monitoring Enabler communicates with probes via middleware which implies additional overhead and latency, i.e. particularly pressing factors when performance is concerned."

The performance measures of CONNECTors in Section 7.3.1 includes a measurement of the additional overhead incurred by instrumented monitoring. It is shown that because communication with the monitoring enabler is asynchronous then only a small additional overhead is incurred.

"According to D3.1, adaptation via learning of a deployed and executed connector is focused on non-functional properties. Could in this process also the "business" functionality of the connector be modified/improved?"

The continuous monitoring and learning of the behaviour of the networked systems may indeed observe that the business functionality of the CONNECTor needs to be modified and indeed this is a case where adaptation of the CONNECTor is to be adapted by the CONNECTability architecture.

1.4 Outline of Report

This report serves two purposes: i) to provide details about the final version of the CONNECT architecture, and ii) to detail work carried out in fourth year of the project. Therefore, the reports follows an outline to present the architecture specification, and emphasize inline where novel work has been carried out.

The CONNECT architecture has been described in detail in previous deliverables, and is composed of the following core elements:

- The CONNECTor architecture specifies the structure of a CONNECTor in terms of the elements that implement and execute it.
- The Networked System Model defines the structure and behaviour of networked systems and served as the common information exchanged between CONNECT enablers. A detailed overview is given in Deliverable D1.3 [3].
- The CONNECT Enabler Architecture defines how the core enablers are deployed and communicate with one another to realise the CONNECT process. The CONNECT Enabler architecture is separated into two distinct phases:
    1. *The Connection architecture*. In this phase, the enablers of the CONNECT architecture identify networked systems to match, and then create the initial CONNECTor between them that is subsequently deployed. The Connection phase is described further in Section 3.
    2. *The* CONNECT*ability architecture*. In this phase, the enablers of the CONNECT architecture monitor the runtime behaviour of the CONNECTor and the connected networked systems in order to detect and adapt to changes that violate the non-functional requirements of the CONNECTor. Also, adaptation is enacted where there is a violation of the functional requirements due to an incorrect definition of the CONNECTor specification. Achieving CONNECTability is detailed further in Section 4.

In Section 2 we describe the final version of the CONNECTor architecture. The goal of the section is to act as a summary of the work carried out over the project towards reaching the final specification; and also to emphasize the further integration work for instrumenting CONNECTors with non-functional behaviour in this final year.

Section 3 presents the first phase of the enabler architecture and presents the complete specification of the enablers' behaviour. The improved integration with the security and DePer enablers in order to analyse the first synthesized version of the CONNECTor with respect to required non-functional properties is reported.

Subsequently, Section 4 discusses the CONNECTability phase of the enabler architecture. In particular this discusses work carried out within the project to investigate complimentary approaches for achieving dynamic adaptation of runtime CONNECTors, and also the management of the running enabler architecture itself.

Ontologies play a central role in the CONNECT architecture. In previous deliverables we have discussed in length about how ontologies are used to address the challenges of interoperability. In Section 5 we discuss the importance of domain ontologies and provide a best practice for developing and using them.

With the widespread adoption of smartphones and related app market, it is now foreseen that the mobile platform will take over the desktop one as the platform of choice for Internet-based applications. In this context, interoperability is significantly challenged due to the diversity of mobile platforms as well as the variety of services being accessed over the Internet for the same purpose. As part of the experimental work carried out within WP6, a significant effort has been devoted to the exploitation and further adaptation of the CONNECT solutions to deal with the interoperability issues that arise in the context of mobile "app" development. Precisely, we have been considering the adoption of CONNECT technologies to support the development of mobile apps that need to interoperate with: (i) other (local or remote) apps or (ii) with remote Internet-based services and in particular cloud-based services. This challenges the CONNECTor architecture given the specific development and deployment processes of mobile apps. The resulting extension of the architecture is sketched in Section 6

The CONNECT architecture is assessed and evaluated in Section 7; this includes an evaluation of the correctness of synthesized CONNECTors with respect to the networked systems requiring them, and an evaluation of the CONNECT results against the state of the art in distributed systems and semantic web.

Finally, in Section 8 we draw conclusions about the progress in year four and identify avenues for future research in this domain.

2 The CONNECTOR Architecture

2.1 Introduction

CONNECTors are generated by the CONNECT process and then deployed between the two networked systems to enable interoperation between them. Therefore, CONNECTors are a fundamental part of the CONNECT architecture. They are both produced and managed by the Enabler architecture (see Section 3) and therefore their properties inform the key architectural principles of the Enabler architecture. They are also directly instrumented in order to achieve CONNECTability properties (i.e. communicating with the enablers that perform monitoring and security see Section 4). Hence, in this chapter we present the final CONNECTor architecture; the core underlying concepts are detailed in this section, whereas the extended features related to specific CONNECT enabler behaviour (e.g. synthesis, monitoring, and security) is discussed further in Sections 3 and 4. We also present the revisions to the CONNECTor architecture (in Section 2.4) with respect to the support of media streaming applications (as initially introduced in Deliverable D1.3 [3]).

The CONNECTor architecture is presented from two viewpoints; a modelling perspective and an implementation perspective:

- The CONNECTors are built and deployed using models generated by the CONNECT synthesis process. We first present k-*Coloured automata* in Section 2.2, these automata concretely describe the behaviour model of a generated CONNECTor.
- Deployed CONNECTors are executed via interpretation; that is, the previously introduced k-coloured automata are interpreted in place. Hence, we secondly present the Starlink tool in Section 2.3; this explains both the implementation and execution of individual CONNECTors.

2.2 k-Coloured Automata

2.2.1 Concrete Models of Networked Systems

CONNECTors, as previously stated, are built upon k-coloured automata. This model stems from the Enhanced Labelled Transition System (eLTS) used to synthesize abstract networked systems and mediators. Hence, the CONNECT architecture provides a way to make the abstract theory concrete in terms of an executable CONNECTor between two physical networked systems.

An *Enhanced Labelled Transition System (eLTS)* (defined in Deliverable D3.3 [13]) representing either the networked systems or the mediator is defined as a tuple $\langle S, Act, \rightarrow, F, s0 \rangle$ where:

- S is a finite set representing the states of the system,
- Act defines the set of observable *actions* that the component requires/provides from its running environment. An *input action* $\alpha = <op,\ i,\ o>$ ($op, i, o \in \mathcal{O}$ where $\mathcal{O}$ is a common domain-specific ontology) requires an operation op for which it produces some input data i and consumes the output data o. Its dual *output action*[1] $\overline{\beta} = <\overline{op},\ i,\ o>$ uses the inputs and produces the corresponding outputs.
- s_0 is the initial state from which the system begins its execution,
- F is the set of states indicating a successful termination of the system, and
- $\rightarrow \subseteq S \times Act \times S$ is the transition relation indicating the change of the system state after performing an action.

A *coloured action automaton* models each networked system as a sequence of actions; these actions match the signature of the operations to be invoked and more precisely the messages exchanged. There are two message types used to model actions: i) a $send\_action$ message which is composed of a set

[1] Note the use of an overline as a convenient shorthand to denote output actions.

```
module PhotoSharing
{
    typedef sequence<string> PhotoMetaDataList;
    typedef sequence<string> PhotoCommentList;

  interface Photo
  {
    // Search for Photos that match a keyword query
    // Return a list of PhotoMetaData describing each picture
    PhotoMetaDataList SEARCHPHOTOS(in string query);

    // Retrieve the URL of the Photo to download
    string DOWNLOADPHOTO(in string PhotoID);

    // Download the comments added to a Photo
    PhotoCommentList DOWNLOADCOMMENT(in string PhotoID);

    // Add a new comment to a given photo
    oneway void COMMENTPHOTO(in string PhotoID, in string comment);
  };
};
```

Figure 2.1: Interface description of the CORBA implemented Photosharing application service

of n output fields such as $field_1 = arg_1, ..., field_n = arg_n$; ii) a $recv\_action$ which is composed a set of n input fields $field_1 = arg_1, ..., field_n = arg_n$. These transitions can be combined to model multiple middleware communication types, e.g., an rpc invocation is modelled as a $send\_action$ message followed immediately by a $recv\_action$ message, whereas a publish-subscribe publication is modelled by a single $send\_action$ message. For example, an interface specification of a photo sharing application is provided in Figure 2.1; here when a client networked system calls the DOWNLOADPHOTO operation it performs a $recv\_action$ message transition (because a mediator, or service receives the sent message) with input fields: $PhotoID = arg_1$ followed by a $send\_action$ message transition with output field $return = arg_1$.

Hence, each networked system is represented as an automaton with edges labeled with messages sent or received according to the signature of the networked system's operations. Figure 2.2 illustrates the coloured automaton for a client networked system implemented in CORBA. In coloured automata, the colour specifies the middleware protocol that is used to execute the requested action transitions (we describe in Section 2.2.3 how colour binding is performed). That is, how $recv_action$s and $send_action$s are concretely executed using a particular protocol. For example, where a $send\_action$ transition is coloured with the CORBA protocol in Figure 2.2 this corresponds to concretely sending a GIOP Request message across a TCP connection. A key advantage of the use of protocol colouring is that the separation of colour from automata transitions allows the same model to be reused and applied to different middleware protocols.

2.2.2 Concrete Modelling of the Mediator

Modelling each of the networked systems is not enough to create an interoperability solution. We must also describe how the translation from one to another is achieved. Therefore, a *mediator* models the interaction between two networked system coloured automata, i.e., it resolves the differences between the two (e.g. in terms of operation sequence mismatch, operation syntax mismatch, and input/output data mismatch) in order that they can interoperate with one another. Hence, a CONNECTor is implemented using such a mediator model.

The model of a mediator is specified using three separate elements: i) the coloured action automaton of NS_1, ii) the coloured action automaton of NS_2, and iii) a model of the translation between the two coloured action automata. Hence, when all three are composed, the mediator can be executed to achieve interoperation. An example mediator specification is shown in Figure 2.3. It can be seen how the two separate coloured automata of the networked systems are merged to form the single mediator.

The mediator specification introduces one important, additional concept: *bi-coloured* nodes (or bridging nodes). These nodes define transitions where the behaviour of one networked system is merged and

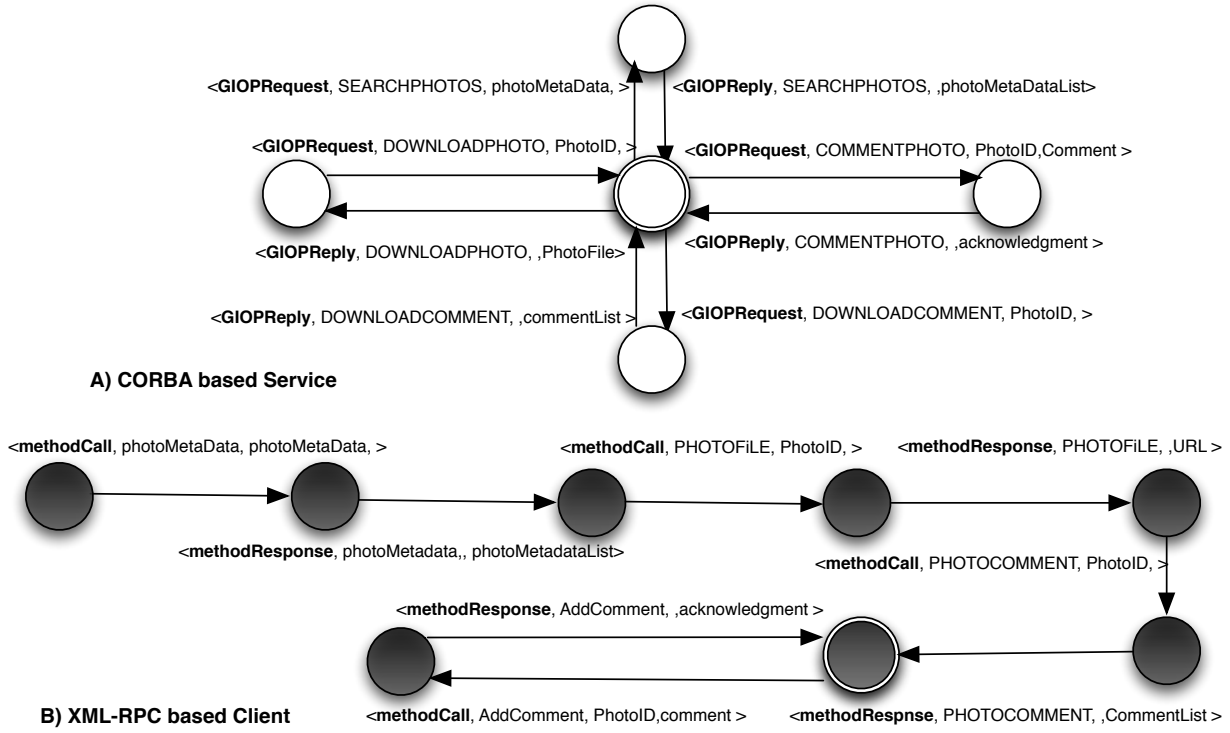


Figure 2.2: Behavioural description of the heterogeneous photo-sharing networked systems

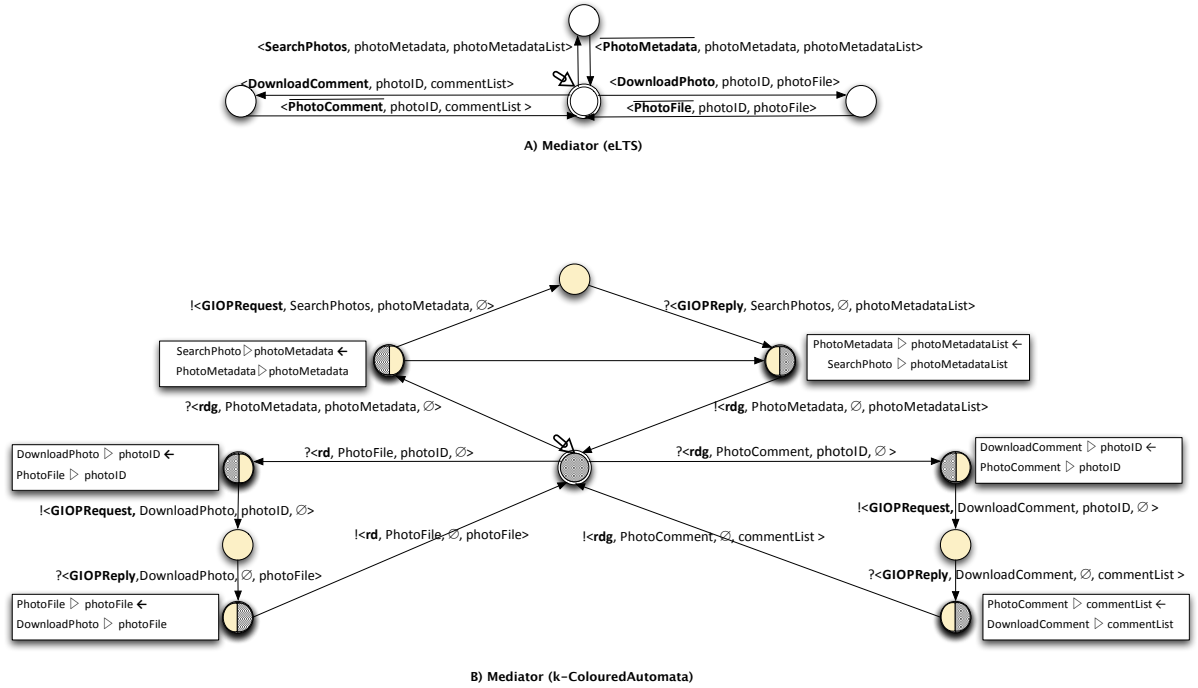


Figure 2.3: Concrete k-coloured automaton of the photosharing mediator

translated with the behaviour of the other networked system. For example, a bridging node provides a transition between the $recv\_action$: DOWNLOADPHOTO transition received from the client and the semantically equivalent $send\_action$: GETPHOTO that is sent to the service. Where bridging takes place it is

not sufficient to simply make a direct translation, e.g. where there are differences in the data parameters. Therefore *bi-coloured* nodes also allow translation logic to be specified, such that when the automaton performs the bridging transition this behaviour will be executed. Translation logic can be specified as follows:

- The value of one input/output field of one action transition can be assigned to the value of an input/output field of another action transition using the *assignment* operator. Constant values can also be assigned.
- The value of one or more input/output fields and constants can be applied to any function and the result can be assigned to value of an input/output field of an action transition using the *function* operator.

2.2.3 Binding Application Action Automaton to Concrete Middleware Protocols

Defining application action automata is the first stage of the mediator process. However, there is insufficient information at this point to execute the mediator, i.e., create an implementation of the CONNECTor. This is because we need to further specify how a $send\_action$ or $recv\_action$ coloured with a middleware protocol is performed concretely in terms of the sending and receiving of specific middleware messages. For this purpose, we introduce *concrete k-coloured automata*.

Middleware protocols differ in the the way they use the network, in terms of the transport protocol used, whether requests are sent by unicast or by multicast, and whether responses are received synchronously or asynchronously. For instance, as illustrated in Figure 2.5(a) and Figure 2.5(b), CORBA and SOAP are both request-response middleware protocols but they differ in message format, port number, network addressing etc. In order to capture these low level network semantics, we again use *colouring* to attach specific middleware protocol information to the concrete transitions of sending a middleware message, or receiving a middleware message. Such information includes: the format of the packet, the connection address, the port number, etc. A complete definition of concrete k-coloured automata is provided in [10].

Abstract Messages. Each transition in a concrete k-coloured automaton represents the sending or receiving of a network message. A network message is organized as a sequence of text lines, or of bits, for a binary protocol, containing both fixed elements (typically found in message headers) and elements specific to a given message (e.g., in the message body). A CONNECTor must extract relevant elements from the received message and use them to create one or more messages according to the target protocols. Similarly, it must extract relevant elements from the received responses and ultimately create a response according to the source protocol. Hence, the k-coloured automata of CONNECTors are based upon these message-based events; and the key design principle is to derive information from network messages and then describe them in a protocol independent manner (such that it can be manipulated as part of the automata specification). We term this protocol independent description the *Abstract Message*. Received network messages are converted to an Abstract Message, correspondingly the Abstract Message is used to build the network message to be sent.

The schema for the Abstract Message content is illustrated in Figure 2.4. This shows that an Abstract Message consists of a set of fields; a field can be either primitive or structured. A *primitive field* is composed of a label naming the field, a type describing the type of the data content, a length defining the length in bits of the field, a boolean stating if this is a mandatory or optional field, and the value of the field, i.e., the data content. A *structured field* is composed of multiple primitive fields. For example, a URL field is composed of four primitive fields: the protocol, the address, the port, and the resource location.

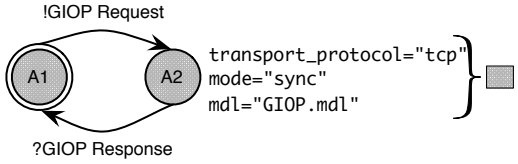
In order to achieve dynamic interoperability, the CONNECTor receives network messages from a networked system (in the format of the protocol employed by this legacy system). This event will trigger the execution of the Mediator, whose behaviour will determine the sequence of actions that manipulate the listeners and actuators. For example, it may receive one or more messages in the Abstract Message format and it may send one or more messages by composing a new Abstract Message and sending this to an Actuator to be delivered to the target networked system.

We now describe the process of binding the application action automata to a concrete k-coloured automaton, which is a concrete specification of the mediator, and hence is the concrete implementation of the CONNECTor. The concrete k-coloured protocol automaton can be directly executed to perform the interaction between two legacy systems.

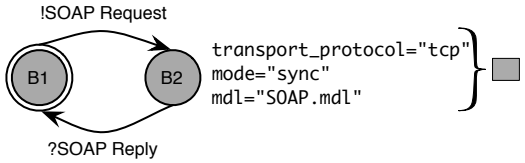
```
<xsd:schema>
   <xsd:element name="Field">
     <xsd:complexType>
         <xsd:sequence>
              <xsd:element name="label" type="xsd:string"/>
              <xsd:element name="length" type="xsd:integer"/>
              <xsd:element name="type" type="xsd:string"/>
              <xsd:element name="mandatory" type="xsd:boolean"/>
              <xsd:element name="value" type="xsd:any"/>
              <xsd:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
     </xsd:complexType>
   </xsd:element>

   <xsd:element name="AbstractMessage">
     <xsd:complexType>
         <xsd:sequence>
              <xsd:element name="Name" type="xsd:string"/>
              <xsd:element ref="Field" minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Figure 2.4: The Abstract Message Schema



(a) IIOP client k-coloured automaton



(b) SOAP client k-coloured automaton

Figure 2.5: Examples of concrete k-coloured automata

To bind to a particular protocol we first require: i) the k-coloured protocol automaton of the middleware protocol to tell us what sequences of messages can perform application actions (e.g. the k-coloured protocol automata for CORBA and SOAP are shown in Figure 2.5) n.b. k-coloured protocol automata were first introduced in Deliverable D1.2 [2], ii) the MDL specification of that protocol's messages (e.g. Figure 2.6) that are used to produce the abstract messages described previously, and iii) the set of rules that describe how a particular protocol (e.g. GIOP) is bound to the application automata concepts (i.e. the action labels, and the parameters).

Figure 2.7 highlights the exact binding procedure. Here the k-coloured application action automaton consists of the sending of an add action followed by the reception of the response. This is then shown to be bound to two concrete middleware protocols (SOAP and IIOP)–this highlights the flexibility of application automaton supporting binding to multiple protocols. First, each action transition is bound to the sequence of messages of the middleware protocol. IIOP and SOAP are both RPC protocols and hence the actions correspond to the request and response messages of each protocol, as seen by the corresponding k-coloured sequence.

Each middleware protocol has a set of *binding rules* (shown at the bottom of the figure) that describe

```
<Message:GIOPRequest>
<Rule:MessageType=0>
<RequestID:32><Response:8>
... <ObjectKeyLength:32><ObjectKey:ObjectKeyLength>
... <OperationLength:32><Operation:OperationLength>
... <align:64><ParameterArray:eof>
<End:Message>

<Message:GIOPReply>
<Rule:MessageType=1>
<RequestID:32><ReplyStatus:32><ContextListLength:32>
...<align:64><ParameterArray:eof>
<End:Message>
```

Figure 2.6: MDL specification of the GIOP message format

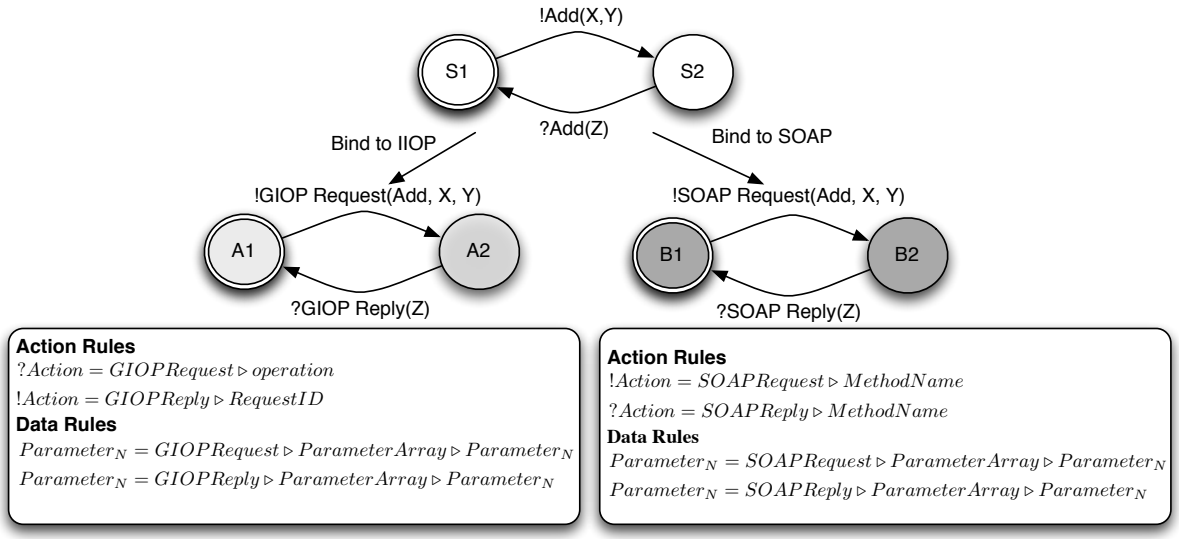


Figure 2.7: Binding to concrete application-middleware automata

how the data in the action transitions are mapped onto the protocol messages. First, the *action rules* state how the action label (e.g. `add`) maps onto the field content within a protocol message described by the protocol's MDL. For example, the `operation` field of the GIOP Request message, and the `methodname` field of the SOAP request envelope. Secondly, the *data rules* state how the data parameters (input and output) map onto the field content of the protocol messages; for example the request action parameters (the X and Y integers) relate to the first two parameters in the ParameterArray field of the GIOP Request message. The return value parameter (the Z integer value) relates to the first parameter of the GIOP reply ParameterArray.

We finally illustrate the application of this binding procedure to the specification of a merged coloured application automaton. When executed the solution resolves both application and middleware heterogeneity. For this example, we continue with the simple addition example. However, in this case NS2 is a SOAP service that provides an add operation with an `int Plus(int, int)` signature whereas the IIOP client (NS1) interface signature is `int Add(int, int)`. Hence, there is application heterogeneity in terms of the operation name. Figure 2.8 shows the result of the merged application automaton when bound to the concrete protocols. On the left side of the figure is the specified application merge, with the bi-coloured states representing the translation of parameters between actions. On the right side is the concrete merged k-coloured automaton, where the action transitions are bound to specific middleware protocols (the operation name difference is overcome by this, after an Add action is received a Plus action is sent). Note, the application translations are bound to the specific MTL translations based upon the previously specified binding rules for SOAP and CORBA (in Figure 2.7).

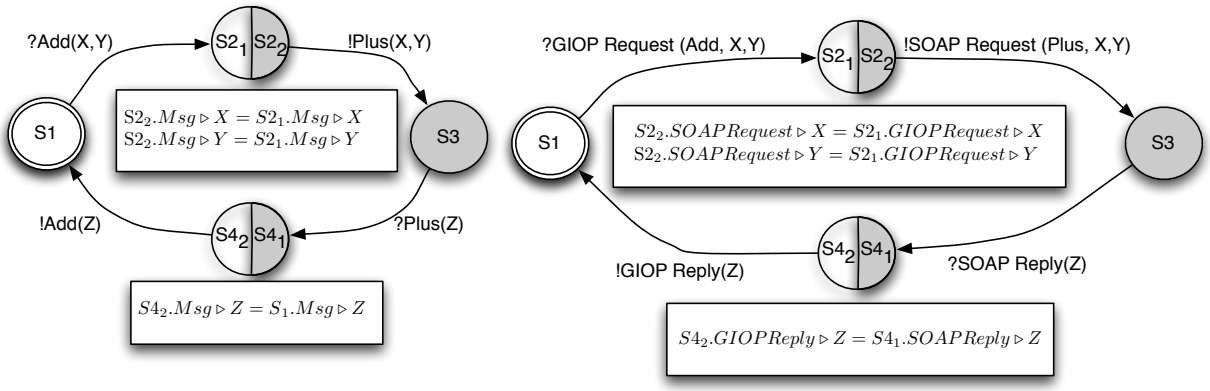


Figure 2.8: Construct a concrete merged application automaton

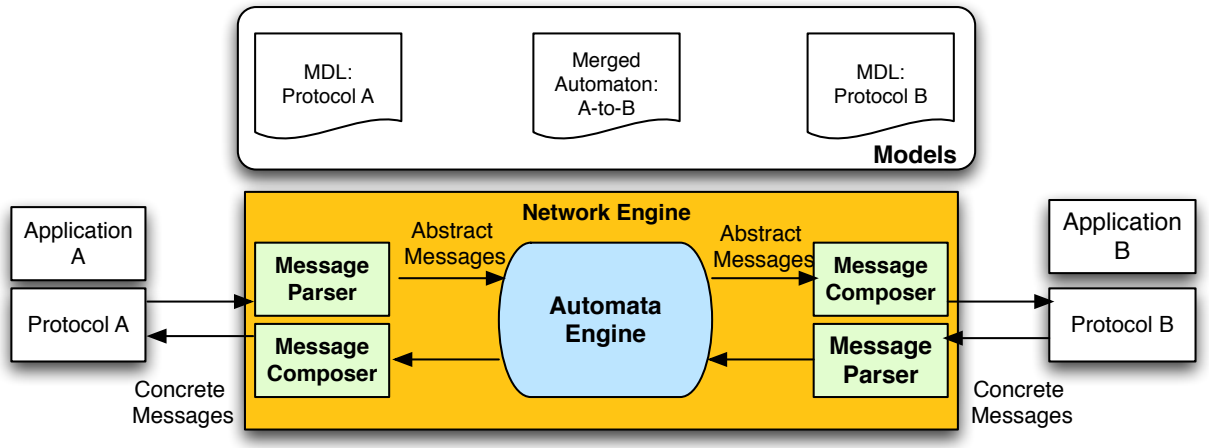


Figure 2.9: Architecture of the Starlink framework

2.3 The Starlink Framework: dynamically interpreting middleware models

2.3.1 CONNECTor Implementation

To deploy and execute a CONNECTor, the previously described concrete models are interpreted and executed using the Starlink tool. That is, Starlink executes on a networked host in the environment; the model is loaded into Starlink, and when the networked systems begin communicating with another Starlink interprets the appropriate transitions in the concrete k-coloured automaton.

A high level vision of a CONNECTor implementation is first presented in Figure 2.9; this illustrates the principle software elements that compose each CONNECTor and their overall behaviour.

- The *Network Engine* provides a library of transport protocols with a common uniform interface to send and receive messages. Hence, it is possible for a CONNECTor to receive messages and send messages using multicast (e.g. IP multicast), broadcast and unicast transport protocols (e.g. UDP and TCP) in order to directly communicate at the network level with the networked systems.

- A *Listener* parses the content of a distinct protocol message (or frame in a streaming protocol). That is, based upon the protocol's message format specification it reads the network data and produces a single *Abstract Message* instance; this is a uniform representation of network messages that is used by the CONNECTor to understand and manipulate the data. *Listeners* are generated from the

description of a single protocol provided as a Message Description Language (MDL) document. For example, the MDL of SOAP messages is used to construct a listener that will parse SOAP messages. A full description of the MDLs developed in the CONNECT project, how listeners are generated, and how listeners execute is provided in Section 4.2 (“Realising Listeners and Actuators”) of Deliverable D1.2 and is not detailed further here.

- An *Actuator* performs the reverse role of a listener, i.e., it composes network messages according to a given middleware protocol, e.g., the SOAP Actuator creates SOAP messages. Actuators receive the Abstract Message as input and translate this into the data packet to be sent on the network via the Network Engines. Like listeners, each actuator is generated from a protocol’s MDL specification.

- The *Automata Engine* forms the central co-ordination element of a generated CONNECTor. Its role is to execute a mediator described as a k-coloured automaton, which documents how content received from one networked system (in the form of an `AbstractMessage`s) is translated into the content and middleware messages required by the other networked system.Hence, the k-coloured automata mediator handles both application and middleware heterogeneity; it is able to address the challenges of: different message content and formats; different middleware protocol behaviour, e.g., sequence of messages; different application data formats; and different application operation behaviour. Each Mediator is specified in terms of k-coloured automata. The automata engine interprets and executes these automata directly.

2.3.2 CONNECTor Instrumentation

Figure 2.10 illustrates an implementation viewpoint of the CONNECTor architecture. Here, the Starlink classes that form a single, complete CONNECTor implementation instance are seen. Here, the diagram shows the Java objects and the method dependencies between them, i.e., a directed arrow is a method invocation (as labelled on the diagram) from one object to the other. This low level representation allows us to illustrate the mechanisms employed to instrument CONNECTors with implementation specific to achieving CONNECTability properties.

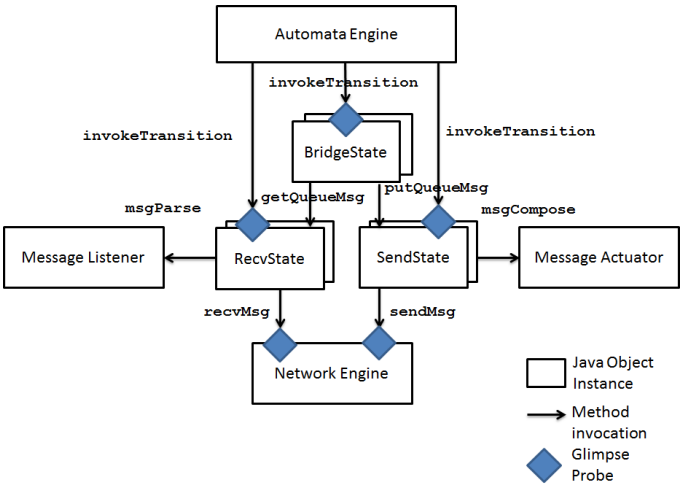


Figure 2.10: Join points where Glimpse probes are inserted into the CONNECTor architecture

In order for a CONNECTor to communicate with other elements of the CONNECT architecture we utilise a simple intrumentation approach. This underpins all of the methods to communicate with the CONNECT processes required to achieve the Connectability properties of monitoring, security and dependability. This approach involves the insertion of *probes* (small pieces of java code) at *join points* within the CONNECTor architecture itself. Where a join point describes a method invocation (or its exception handler) with the CONNECTor implementation objects (seen in Figure 2.10).

There are two types of probe:

- *A Glimpse probe*. An example of the code for this probe is shown in Figure 2.10. This constructs an event using the information from the method invocation, i.e., that this is an `invokeTransition` and hence an there has been a state transition in the automaton. The message is then sent to the Glimpse monitoring bus so that it can be received by the monitoring enabler.
- *A Security Probe*. This is a similar piece of code to the Glimpse probe. It is similarly inserted at join points in the CONNECTor architecture. This code will be produced in the final year of the project, and then integrated into the running CONNECTors.

Figure 2.10 demonstrates the join points in the CONNECTor architecture where the Glimpse probes are inserted. Note insertion means the code is embedded into the source of the actual method (i.e. it is an invasive insertion). These methods are the three `invokeTransition` methods of the different state type objects. Hence, the probes monitor when the state transitions occur. Further probes are inserted into the Network Engine object on the send and receive methods in order that the physical messages can also be monitored.

2.4 CONNECTors for Media Streaming

In Deliverable D1.3 [3] we introduced AmbiStream, a fully-distributed CONNECTor architecture, for achieving Mobile Interoperable Live Streaming. We also integrated the AmbiStream CONNECTor with iBICOOP middleware, which implements Discovery and Communication Enablers for mobile environments.

The AmbiStream mobile CONNECTor was designed to solve interoperability between live streaming protocols on two levels: at the Control Protocol level and at the Media Container Format level. In this context, streaming protocol interoperability is possible when the following assumptions are satisfied: (i) Both the Source and the Destination support a common pair of audio/video codecs and (ii) the codec pair used is compatible with the Destinations (client-side supported) streaming protocol.

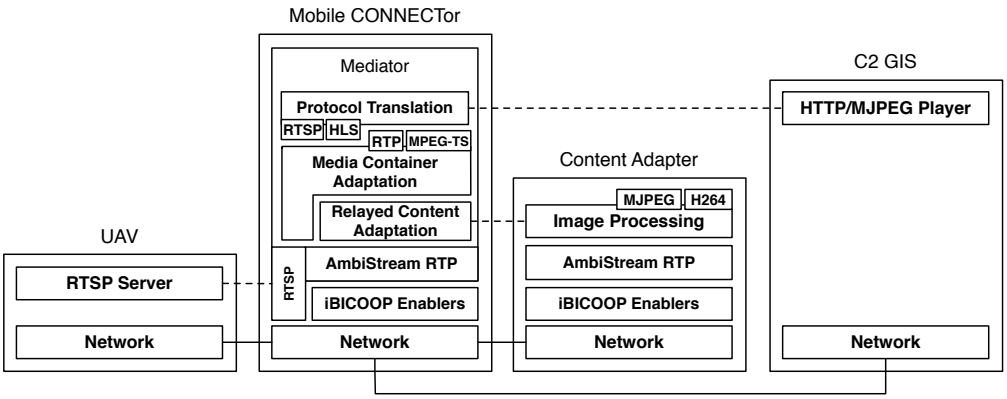


Figure 2.11: The AmbiStream CONNECTor architecture

This year, we further extended the AmbiStream CONNECTor architecture in order to reduce the upnoted assumptions. The extensions are illustrated in Figure 2.11. This is achieved by adding a Relayed Content Adaptation layer to the mobile CONNECTor architecture. Since low-level signal processing (i.e., image compression) is resource intensive, we deployed the Content Adaptation unit as a cloud service on the Internet. In order to allow CONNECTor mobility, the Content Adapter was designed on top of the iBICOOP middleware, which provides nomadic mobility support (i.e., seamless vertical and horizontal network hand-off support). While the Content Adaptation is done remotely, the mediation is still conducted locally on a mobile device. When required, the Content Adaptation is achieved by sending the Elementary-Stream content obtained from the streaming Source to the remote Content Adaptation Service, using AmbiStream RTP as multimedia transport protocol. The stream is then transcoded by the Content Adapter and relayed back to the Mobile CONNECTor.

2.5 Summary

During the course of the CONNECT project the CONNECTor architecture has been refined to handle increasingly complex interoperability challenges. The final stable version presented in this chapter has been proven to successfully manage differences in application behaviour, middleware protocols, application data, and non-functional concerns. In Year 4 of the project we have specifically focused on finalizing the instrumentation of CONNECTors, and further details of how this has been realised with Security is provided in Section 3.2.4. Additionally, the focus in Year 4 has been on the evaluation of the CONNECTors. Section 7 presents a number of cases involving a broad range of heterogeneity in terms of application behaviour, application data, and middleware protocols and it is shown that the CONNECTor architecture can be successfully employed in every case to achieve interoperability.

3 The Connection Architecture

3.1 Introduction

As previously described in Section 1.4, the overall CONNECT behaviour and functionality is captured by the *Enabler Architecture*. In this section of the final CONNECT architecture we present the CONNECTion phase of the Enabler architecture, which we term the CONNECTion Architecture. This is the first phase of the CONNECT process and performs the task of discovering networked systems, and then matching the ones to interoperate with one another; subsequently the initial CONNECTor is synthesized and deployed such that the networked systems can interoperate at a functional level.

The CONNECTion architecture was initially specified at the end of the third year of the project. In this fourth year, work has focused on further implementation improvements and refinements to the specific enabler interactions within this architecture; in particular, the key refinements involve how the Security and DePer enabler receive the synthesized CONNECTor, analyse the provided model and then instrument it according to the required CONNECTability properties (see Section 3.2.4). Hence, in this section we reintroduce the architecture from deliverable D1.3 [3] and add details of the extensions from the fourth year work.

3.2 Architecture Specification

The configuration of the enablers to perform connection is illustrated in Figure 3.1. Here, software components are represented using the UML Component diagram notation (each enabler is a single instance of a software component). The interfaces provided by these components are represented by a line ending with a filled circle. Receptacles are represented by lines ending with a semi-circle arc and these represent the services required by a software component. A join of an interface and a receptacle defines a message-exchange binding, i.e, enablers communicate by sending messages to one another; bindings can be one of two styles: i) a request-response exchange where the request message contains input parameters and the response message contains the result data, ii) a message notification where a message is sent asynchronously and there is no response.

Table 3.1 briefly describes each of the enablers (which are at present all singleton components, i.e. only one instance of each operates in the network and as part of the overall CONNECT process). The final progress of the implementation is stated, and the table also provides references to further details about enabler implementations from prior deliverables.

To explain the behaviour of the Enabler architecture in order to achieve CONNECTion, the sequence of messages exchanged by enablers (via their external interfaces) is shown in Figure 3.2. Here the diagram follows the UML message sequence diagram notation (the legend in the figure defines the style of message exchange depicted by the notation).

3.2.1 Discovery Enabler: Discovery of Networked Systems

In Figure 3.2 the *Discovery enabler* is the starting point for behaviour in the CONNECTion phase. Each networked system is discovered irrespective of the discovery protocol that is used for advertisement. When a networked system is discovered this produces the internal event (`DiscoverNS`); where an internal event is an occurrence internal to a single enabler that triggers external events exchanged via the enabler's interfaces. This internal event of the Discovery enabler initiates the behaviour to build a Networked System Model that provides richer information about the system than is provided by the discovery protocol. The first task is to infer, if necessary, the affordance and to learn the behaviour of this networked system, and so the discovery enabler send a `learnCNSBehaviour` message to the Learning enabler's `LearnNS` interface (the message contains the identifier of the networked system to learn as the parameter).

3.2.2 Learning the Networked Systems

The Learning enabler then retrieves the interface of the networked system (by sending the `getNSInterface` message to the `NSRepository` interface of the Discovery enabler) and uses this

Table 3.1: Implementation and integration progress of CONNECT enablers for Connection

Enabler	Implementation Status	Integration Status
Discovery	Complete Deliverable D1.3 [3] Section 4.1	Singleton component. Discovers networked systems in face of discovery protocol heterogeneity. In collaboration with the learning enabler builds Networked System Models for each system, learns affordances, and matches networked systems based upon goals and intent. **Fully implemented and integrated.**
Learning	Complete Deliverable D4.4 [16]	Singleton component. The Learning Enabler uses active learning algorithms to dynamically determine the interaction behaviour of a networked system from the initial CNSInstance representation and produces a model of this behaviour in the form of a labeled transition system (LTS) **Fully implemented and integrated.**
Interaction	Complete Deliverable D1.3 [3] Section 3.2	Singleton component. Uses the Starlink tool to dynamically invoke networked system actions irrespective of the middleware protocols employed. **Fully implemented and integrated.**
Deployment	Complete	Singleton component. Receives k-coloured automata from the synthesis enabler, and deploys this on a running instance of the Starlink tool; this can be on the same host as the Deployment enabler, or on a separate identified host within the network. **Fully implemented and integrated.**
Synthesis	Complete Deliverable D3.4 [18]	Singleton component. Takes the enhanced LTS models of a pair of networked systems and constructs a CONNECTor model in the form of a k-Coloured automaton. It then calls both the Security and DePer enablers to analyse the CONNECTor. These return instrumented versions of the CONNECTor which the synthesis enabler can deploy. **Fully implemented and integrated.**
DePer	Complete Deliverable D5.4 [15]	Singleton component. Receives a k-coloured automaton from the synthesis enabler and analyses the CONNECTor against the dependability properties discovered about the networked systems. Identifies if the CONNECTor maintains these properties, and suggests changes to the CONNECTor otherwise. **Implemented and integrated.**
Security	Complete Deliverable D5.4 [15]	Singleton component. Receives a k-coloured automaton from the synthesis enabler and based upon the security policies discovered about the networked systems, instruments a new CONNECTor that enforces the security policies using security probes. **Fully implemented and integrated.**
Trust	Conceptually complete Deliverable D5.4 [15]	Singleton component. Analyses and monitors a CONNECTor to determine if the level of trust provided by networked systems meets the trust requirements. To do this, the trust enabler analyses the model provided by the Synthesis enabler. **Implementation not integrated.**

to build a richer model of this networked system's behaviour. In order to do this it must interact directly with the networked system being learned, therefore it uses the Interaction enabler (sending an `invoke` message of the `Invoke` interface); here the abstract content describing the required invocation is mapped to the concrete legacy protocol used by the NS and the results of the invocation are returned in a message to the Learning enabler. Once learning is complete the NSModel is updated and then placed in the Discovery enabler repository by sending the `updateNSBehaviour` message to the `NSRepository` interface.

Inria, TUDO and the university of Trento (from EternalS co-ordination action) collaborated to further investigate the central role of advanced learning techniques in supporting the concept of Emergent Middleware. Besides using automata learning to determine the behaviour of a networked system (as defined

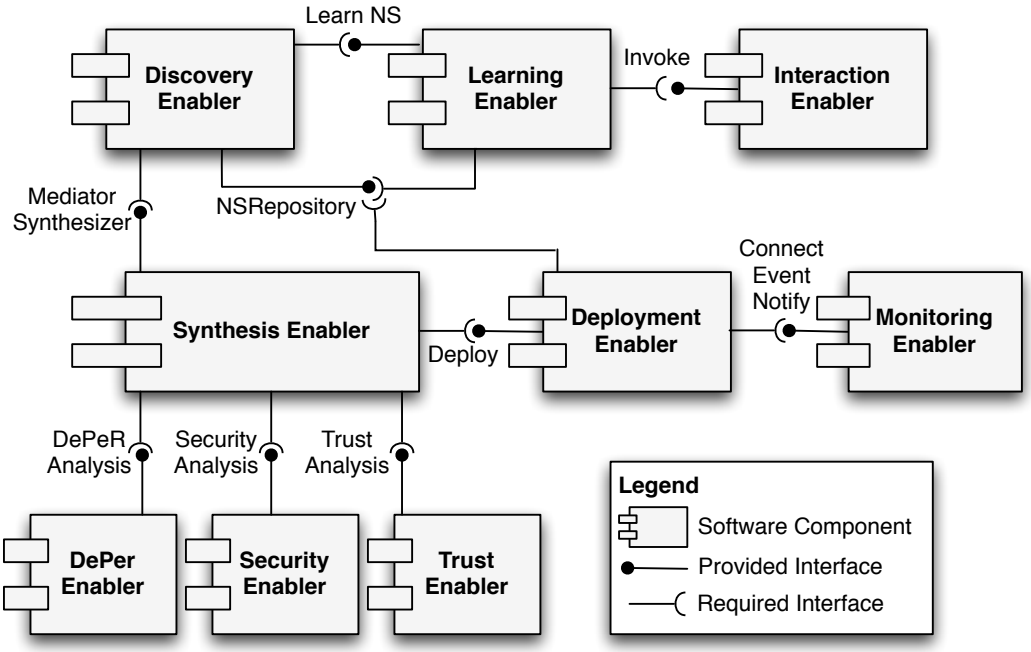


Figure 3.1: The configuration of the enabler architecture for the connection phase.

in WP4), we exploit statistical learning to enable on-the-fly inference of functional semantics of NS in terms of the ontological annotations of the affordance and interface operations and data. This work is presented in paper P3 in the appendix.

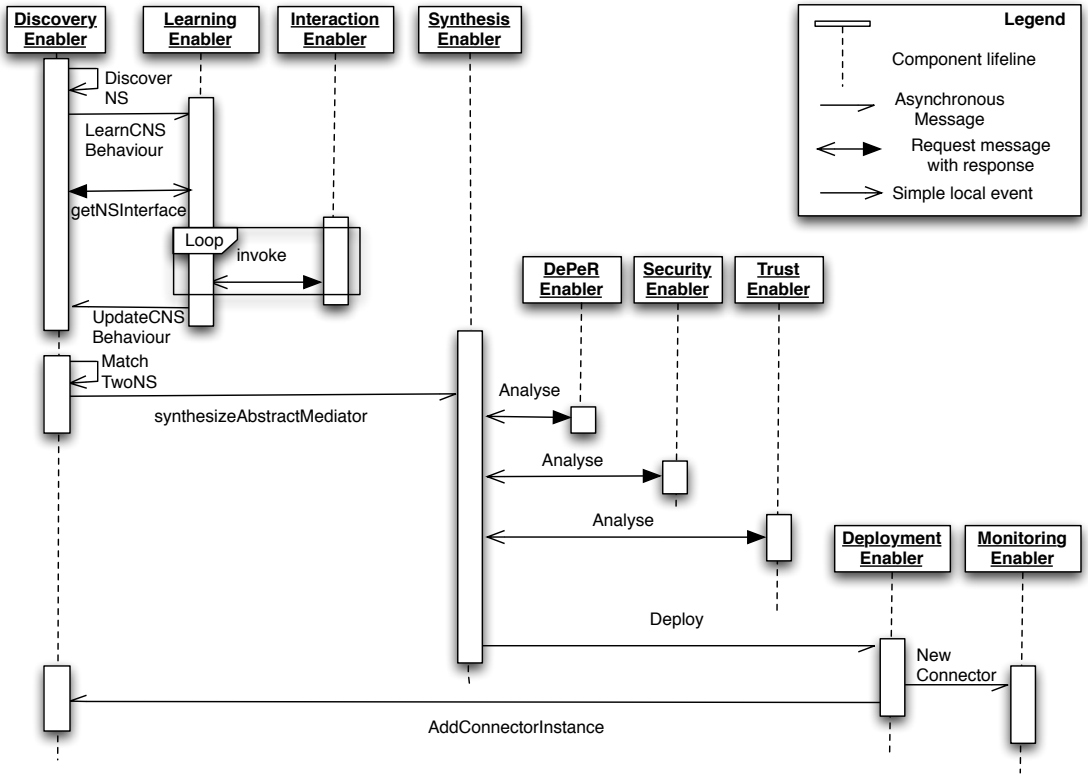


Figure 3.2: The sequence of messages exchanged by CONNECT enablers for connection.

3.2.3 Matching Networked Systems and Creating the CONNECTor

The Discovery enabler matches networked systems and decides where an interoperability solution is required. The result of this matching process is an internal event (`MatchTwoNS`) that initiates the construction of a CONNECTor between two networked systems. Here, the Discovery enabler sends the `synthesizeAbstractMediator` message to the `MediatorSynthesizer` interface (passing the models and information about the NS as parameters). The Synthesis enabler then constructs the initial k-Coloured automaton of the mediator in order for it to be analyzed against the non-functional requirements. Subsequently, an `Analyse` message is sent to each of the three connectability enablers (i.e. the `DePerAnalysis`, `SecurityAnalysis`, and `TrustAnalysis` interfaces) and any potential refinements are returned as changes to this CONNECTor model. The CONNECTor is then deployed when the Synthesis enabler sends the `deploy` message to the Deployment enabler's `Deploy` interface. This creates and deploys the concrete CONNECTor specification and an instance of the Starlink interpreter [10] to execute the specification as was described in the previous section.

Once the CONNECTor is initially deployed this is the end of the CONNECTion phase. At this point, monitoring begins and the CONNECTability phase of the architecture is started, which we discuss fully in Section 4.

3.2.4 Creating an Instrumented Security Policy-based CONNECTor

We now provide more detail about the behaviour when the Synthesis enabler calls `Analyse` behaviour on the Security Enabler. Security is enforced based upon policies associated with a particular networked system. For example, the operations to fly a drone may include a `moveup` command; a security policy may state that the drone may not fly above a certain height, and hence when we produce a CONNECTor we must ensure that this policy is maintained. The security policy is discovered alongside a networked system and is passed by the Synthesis enabler along with the three xml documents describing the current concrete coloured automaton of the mediator. The security enabler then extracts which networked system the policy is for (i.e. either the client side or service side) and from that point works with only the XML file of the concrete coloured automaton of that networked system. The operation `generateSecureXML` as listed below is then performed.

```
1  public interface Security{
2    /**
3    * Construct a connector and deploy, deposit the ConnectorInstance into the repository
4    * and return the ID of this.
5    * @param NSColouredAutomaton The XML model of the concrete coloured automaton.
6    * @param securityPolicy The security policy for this networked system.
7    * @return The new instrumented version of the coloured automaton in XML.
8    */
9    public String generateSecureXML(String NSColouredAutomaton, String securityPolicy);
10 }
```

The initial concrete coloured automaton may contain states and transitions as follows. This states that when a moveup operation is sent to networked system (service side i.e. the actual drone) the authentication session token and height to move up are passed as parameters. Once performed the transition moves to Drone20 which receives the response from the service:

```
1  <state>
2    <label>Drone19</label>
3      <transition>
4        <action>moveup</action>
5        <operation>send</operation>
6        <Outputs>
7          <Output>
8              <simpleType>
9                <name>accesstoken</name>
10               <type>java.lang.string</type>
11             </simpleType>
12         </Output>
13         <Output>
14             <simpleType>
15               <name>height</name>
```

```
16                  <type>java.lang.double</type>
17                </simpleType>
18              </Output>
19            </Outputs>
20            <toState>Drone20</toState>
21        </transition>
22    </state>
```

The `generateSecurityXML` operation produces a new coloured automata as follows. Here a guard tag is attached to the `moveup` transition; when the `moveup` transition is performed the policy (the value of the policy tag) is checked. Section 4.2 describes how instrumented probes within the CONNECTor remotely call the Security enabler in order to manage this behaviour. If the guard returns true the automata behaves as normal (i.e. moves to state `Drone20`, however where the security policy is validated the `<exceptionstate>` tag informs a new transition that must be taken. In this case it goes to state Drone19_S which is a noaction state, and hence no operation is called on the service and the automaton moves to state Drone21 which is the position after the response would have been received. Therefore, the policy detected that the drone should not go up, and bypassed the operation that would have made it do so.

```
1  <state>
2    <label>Drone19</label>
3    <transition>
4      <action>moveup</action>
5      <operation>send</operation>
6      <guard>
7          <policy> ... </policy>
8          <exceptionstate>Drone19_S</exceptionstate>
9      </guard>
10     <Outputs>
11       <Output>
12         <simpleType>
13           <name>accesstoken</name>
14           <type>java.lang.string</type>
15         </simpleType>
16       </Output>
17       <Output>
18         <simpleType>
19           <name>distance</name>
20           <type>java.lang.double</type>
21         </simpleType>
22       </Output>
23     </Outputs>
24     <toState>Drone20</toState>
25   </transition>
26 </state>
27
28 <state>
29   <label>Drone19_S</label>
30   <transition>
31     <action>moveup</action>
32     <operation>noaction</operation>
33     <toState>Drone21</toState>
34   </transition>
35 </state>
```

3.3 Summary

In the fourth year of the project the CONNECTion phase of the architecture has been refined and the implementation completed. Much of the implementation was already in place from the prior year, however significant progress was made towards the final implementation of the Security and DePer functionality within this CONNECTion phase. This involved making changes to the Starlink tool in order that it could understand and interpret Security features and in turn make use of security probes in order to monitor behavior against the defined policies. Hence, functional CONNECTors that resolve the interoperability

challenges can be created and deployed, but also CONNECTors that respect non-functional properties can also be constructed, interpreted and executed within the CONNECT architecture.

4 The Connectability Architecture

4.1 Introduction

The second phase of the CONNECT architecture follows the creation of an initial CONNECTor solution and manages the continued operation of the interoperability solution; that is, it supports eternal interoperability. The CONNECTability architecture provides this behaviour; in particular, the CONNECTability enablers monitor the runtime behaviour of both the produced CONNECTors and the connected networked systems in order to detect and adapt to changes that violate the non-functional requirements of the CONNECTor.

In this chapter we first present the details of the CONNECTability architecture and how it behaves. We identify that the integration of dynamic adaptation of CONNECTors remains conceptual; however, we then present collaborative work that investigates approaches to achieve dynamic adaptation. This work has been published as two separate papers, which we include as an appendix and reference in this section.

4.2 Realising Connectability

Once a CONNECTor is deployed and executing, the configuration of enablers as seen in Figure 4.1 behaves so as to achieve the required CONNECTability properties of the CONNECTor. Hence, this configuration concentrates on monitoring the behaviour of the networked systems and CONNECTors and dynamically adapts the CONNECTor when the non-functional requirements are no longer satisfied. Importantly, software elements are inserted into the CONNECTor (as described in Section 2.3.2) in order for CONNECTors to communicate with the enablers directly.

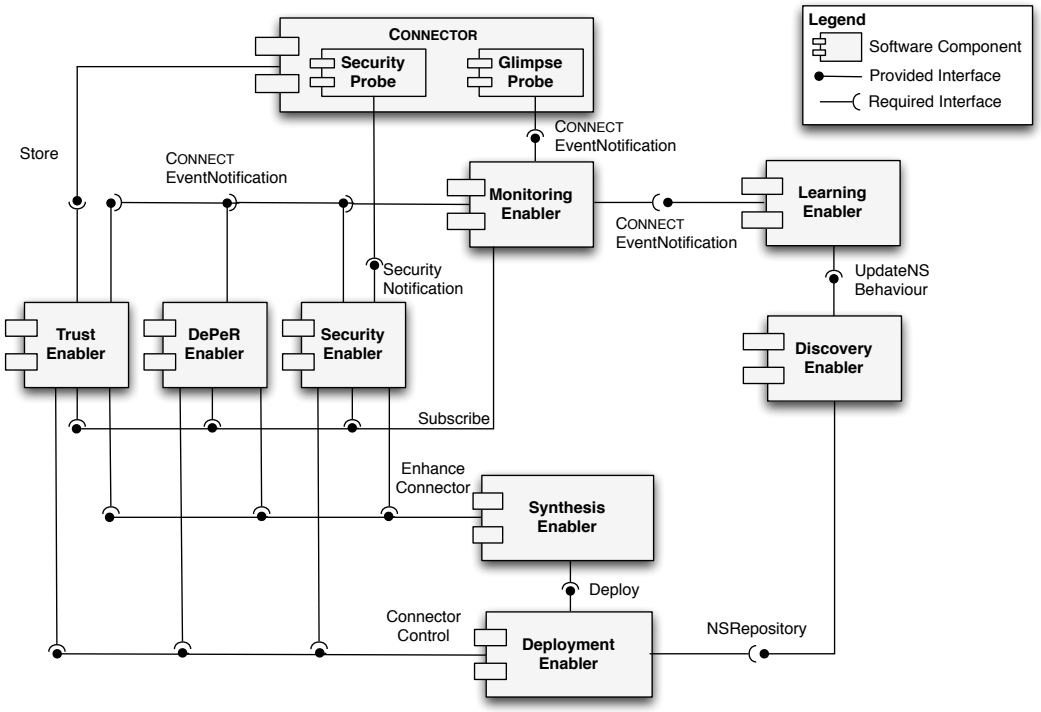


Figure 4.1: The configuration of the enabler architecture for the connectability phase.

Once again we use message sequence diagrams to highlight how the architecture is employed to achieve the dynamic monitoring and adaptation of CONNECTors. Figure 4.2 demonstrates the behaviour performed by the Monitoring and DePer enabler as part of the Connectability phase. The DePer enabler first subscribes to be notified of events of interest that are generated by the CONNECTor. Here, DePer

sends a `subscribe` message (with the subscription filter as a parameter) to the `Subscribe` interface of the Monitoring Enabler.

All events generated by the CONNECTor are sent by the Glimpse probes embedded within the CONNECTor to the Monitoring enabler component via the `Publish` message of the `ConnectEventNotify` interface. Where these match a required subscription (of the DePer enabler) the Monitoring enabler then forwards the event message via the `GlimpseBaseEvent` message to the individual enabler.

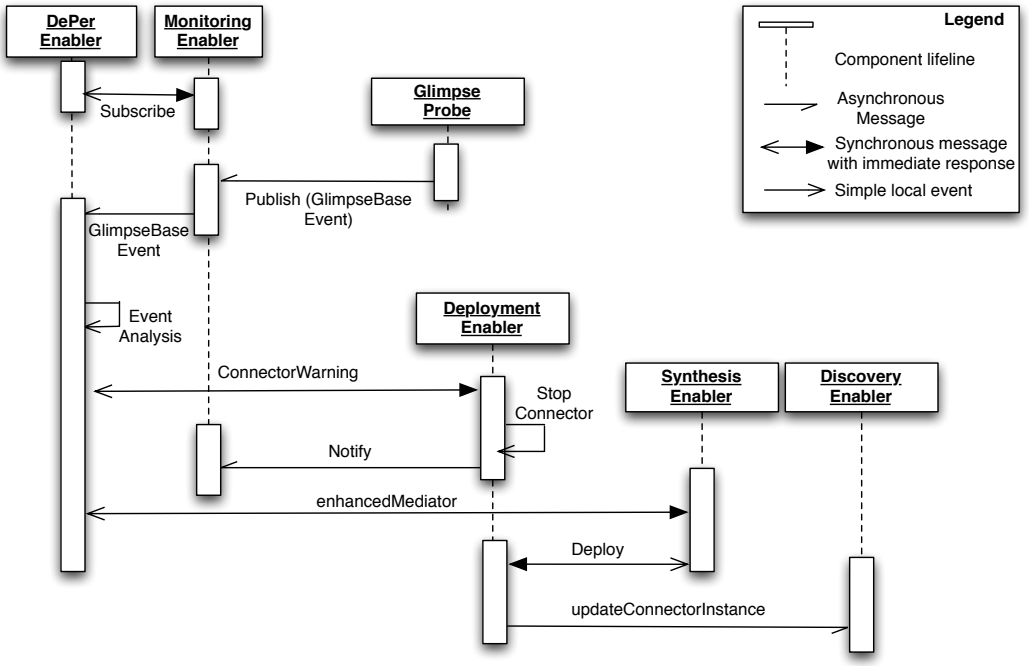


Figure 4.2: The sequence behaviour of monitoring and dependability adaptation

The DePer enabler monitors and analyses the behaviour of the CONNECTor with respect to the non-functional requirements, starting the adaptation process where there is a violation of those requirements. Hence, Figure 4.2 illustrates the sequence of events where the dependability enabler analyses the incoming CONNECTor events. When it detects a violation of the non-functional requirements it first sends a `ConnectorWarning` message to the `ConnectorControl` interface of the Deployment enabler. The Deployment enabler then performs a local `StopConnector` operation in order to place the CONNECTor in a safe state to adapt. Subsequently it sends an `enhancedMediator` message to the Synthesis enabler. The Synthesis enabler then constructs the new concrete CONNECTor based upon this model and then creates a new instance of the CONNECTor by sending the `deploy` message to the Deployment enabler. Finally, the CONNECT repository (which is part of the Discovery enabler) is updated with the new information about the CONNECTor instance when the Deployment enabler sends the `updateConnectorInstance` message to the Discovery enabler.

A similar adaptation process is enacted by both the Security and Trust enablers (as seen in Figures 4.3 and 4.4). This describes the planned integration of the to be completed Security and Trust enablers. When a CONNECTor executes a guard transition, i.e., there is a security policy associated with the transition, the security probe embedded in the CONNECTor calls the Security Enabler directly (this is implemented as a Web Service call via SOAP). The Security Enabler checks the policy based upon the parameter data received and the current context information captured about the system and returns a result stating if the policy is violated (which is used by the Starlink interpreter to determine how to execute the instrumented automaton-as described in Section 3.2.4). When a security violation occurs the probe then sends this information to the monitoring enabler. These violation events are used by the Security enabler to decide if it should reinstrument the CONNECTor using the adaptation loop.

The Trust enabler is more proactive: it continuously monitors the trust of a networked system by using the Interaction enabler to test the provided service of each networked system. Based upon the

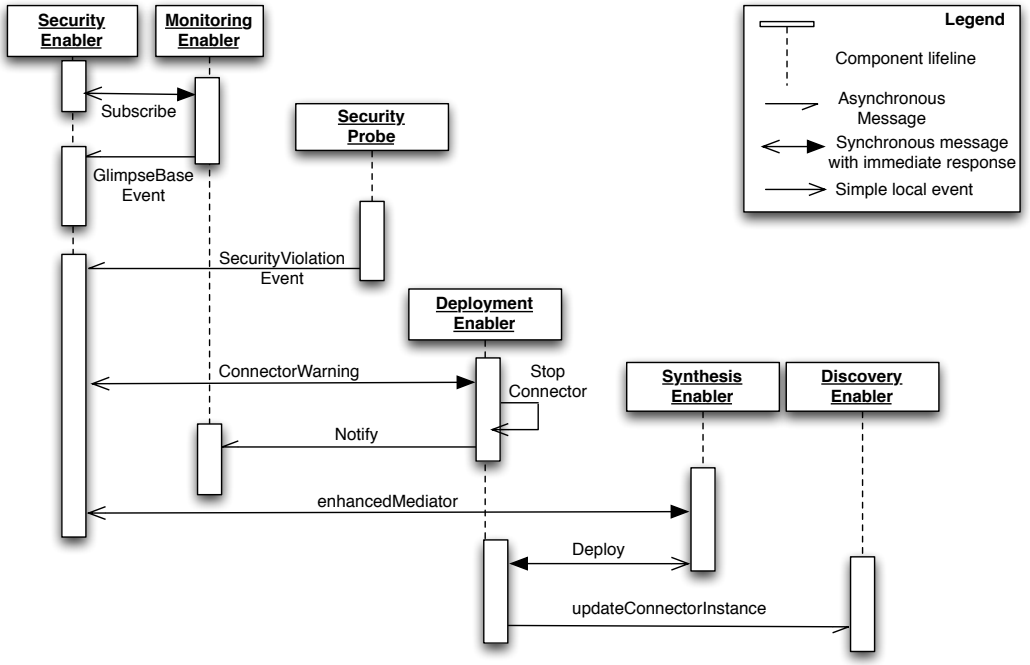


Figure 4.3: The sequence behaviour of monitoring and security adaptation

result of this it calculates a trust value for the CONNECTor and stores this information as meta-data within the CONNECTor (sending the `PutMetaData` message to the CONNECTor's `Store` interface). Note, the current trust value is read by sending the `GetMetaData` message to the `Store` interface Once again, when there is a violation of trust the adaptation mechanism to stop, resynthesise and redeploy is initiated by the Trust Enabler.

As with CONNECTion, we list the progress details of each of the enablers in terms of their achieving the previously described behaviour by their implementations. Table 4.1 demonstrates the progress.

4.3 Adaptation in the CONNECT Architecture

4.3.1 introduction

Table 4.1 illustrates that realised adaptation within the CONNECT architecture remains conceptual. There is no concrete implementation of the underlying concepts to illustrate the potential that adaptation has in supporting long-lived CONNECTors and solutions that are deployed in highly dynamic environments. Hence, within WP1 two pieces of individual work has been carried out by WP1, these explore different solutions based on model-based adaptation and offer a roadmap for future adaptation in the CONNECT architecture:

- *Models@runtime in* CONNECT. Inria and Lancaster collaborated to explore the potential of CONNECT models themselves to underpin future adaptation work, e.g., the dynamic generation of models, and the regeneration of new models in the face of change. This work is presented in the paper P1 attached in the Appendix.

- *SM@RT tool adaptation of k-coloured automata*. PKU and Lancaster collaborated to explore how the SM@RT tool developed at PKU can be used to concretely realise adaptations within a CONNECTor. This work is presented in the paper P2 attached in the Appendix.

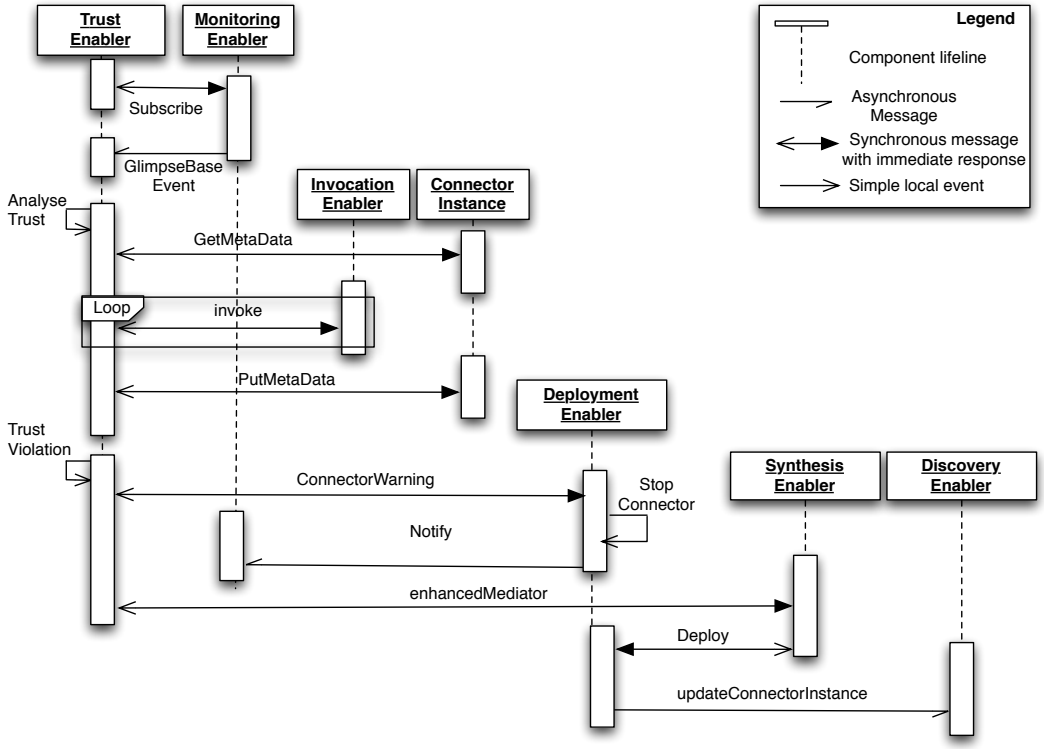


Figure 4.4: The sequence behaviour of monitoring and trust adaptation

Table 4.1: Implementation and integration progress of CONNECT enablers for Connection

Enabler	Implementation Status	Integration Status
Monitoring	Complete Deliverable D5.3 [12]	Singleton component. Receives monitoring data from probes instrumented in the CONNECTor. Notfies information to channels that other enabler's subscribe to **Fully implemented and integrated.**
Learning	Complete	Singleton component. The Learning Enabler receives continuous monitoring data in order perform active learning about the functioning CONNECTor; where new behaviour is learned the models in the discovery enabler are updated **Fully implemented and integrated.**
Security	Partial Deliverable D5.4 [15]	Singleton component. Security probes notify this enabler of policy to test, and where violations occur. This behaviour is **Fully implemented and integrated.** Security based adaptation not implemented.
Trust	Not included	Trust-based monitoring and adaptation loop not integrated.
DePer	Partial Deliverable D5.4 [15]	Singleton component. Receives data from the monitoring enabler to continuously analyse the dependability and performance metrics of the CONNECTor against the non-functional requirements. Leverages this information to inform adaptation. Partial implementation complete, not fully integrated.

4.3.2 Management and Adaptation of the Enabler Architecture

Introducting the SM@RT tool

In CONNECT, the emergent middleware enabling interoperability between heterogeneous networked systems, i.e. the CONNECTor, is generated at runtime and may be deployed in highly dynamic network environments. The CONNECT Enabler Architecture is the underlying system realizing the goals proposed by CONNECT. As previously discussed this architecture consists of two parts: enablers and channels. Enablers are the functional software entities, executing to focus on specific tasks. Channels are the communication medium between enablers, making it possible for enablers to collaborate. From the perspective of implementation and deployment, enablers are wrapped as OSGi bundles working independently as autonomous entities, while channels are provided by Java Messaging Service (JMS) which is a message-oriented distributed software solution. Just like CONNECTors, enablers and channels are also deployed in dynamic network environment. As a result, the CONNECT Enabler Architecture itself has to be adaptable to changing environment. In other words, the enablers and channels themselves should be managed at runtime.

We utilize SM@RT to help address the management issue of the Enabler Architecture. SM@RT, i.e. Supporting Model AT Run Time [26] [31] [35] [33] [32], consists of a domain-specific modeling language (called SM@RT language) and a code generator (called SM@RT generator) to support model-based runtime system management. By specifying what elements to be managed and how to manipulate them, SM@RT generates the synchronization between the runtime model and corresponding systems. We regard the CONNECT Enabler Architecture as the target system to be managed and employ SM@RT to make the architecture adaptable to runtime changes.

Adapting the Enabler Architecture

Figure 4.5 shows the meta-model of the CONNECT Enabler Architecture from the runtime view. This meta-model specifies what elements have to be managed by the administrator. It can be simply divided into two layers:

- The top part is the enablers whose role is to implement specific functionalities. There are nine kinds of enablers. All of them are wrapped as OSGi bundles and deployed by OSGi Server. Therefore, we build a class ”OSGiServer” representing the container where all the enabler bundles are located. The combination relationship between OSGiServer and each kind of Enabler means that the enabler can be managed by the server. Note that the multiplicity of enabler class is ”1..*”, indicating that there may be more than one instance of each enabler. This proves useful for load balancing of the enablers.

- The bottom part is the channels whose role is to enable communication between enablers. Each enabler provides some interfaces for other enablers to use the functions. Some interfaces are wrapped by Connect JMS wrapper while others related with monitoring are supplied by GLIMPSE.

Therefore, there are two server classes concerning with channels: `ConnectJMSWrapperServer` class manages the channels which are implemented by Connect JMS Wrapper; GlimpseServer is responsible for `ConnectEventChannel` and `GlimpseConsumerChannel`. It should be pointed out that there is only one instance of each channel class except the `GlimpseConsumerChannel`. This constraint simplifies the implementation of enablers without loss of efficiency.

The relationship between enabler class and channel class models the functional dependencies between enablers. Each enabler provides interfaces for others to invoke. So both the enabler providing interfaces and those requiring the interfaces have an association with the specific channel class. All the enablers associated with a specific channel play the role of both publisher and subscriber. However, for one channel class, there is only one enabler class receiving request messages and publishing response messages, while other enabler classes publish request messages and receive response messages. The former enabler class is who provides the specific interface with respect to the channel class. This can be easily known from the target role name of the association relationship.

Given the meta-model, the next step is to specify how to access each element for management. This can be simply solved thanks to the underlying techniques we choose. For enabler classes, since they are

OSGi bundles, the OSGi framework provides management API to install, start, stop and update bundles as well as to acquire runtime information. For channel classes, since they are implemented by JMS, there are also management APIs provided by JMS such as collecting the number of messages delivered to topics in given period. Based on the APIs, SM@RT is able to generate the synchronization engine: any architecture's changes can be reflected to the runtime model and any changes to the model can be applied to the architecture on the fly.

With the help of SM@RT, it is simple to make the enabler architecture adaptable to changing environment, since model-based techniques (like QVT and OCL) can be used to facilitate management. We give some examples as below.

1. Monitor the location and running status of each enabler, e.g.
    - Latency of each request's process. For the channel we monitor, record the time interval between the request message and its corresponding response message.
    - Throughput of enabler. Count how many requests are processed by the monitored enabler.
    - Network environment of the channel, such as the latency, the publishers and subscribers to the channel.
2. Adapt to changing environment, e.g.
    - Load balance by starting or killing instances of enabler and scheduling the requests to enabler.
    - Fault tolerance by replication. For each kind of enablers, start more instances and they process the same requests.
    - Redeploy the enablers or channels to a better network.

4.4 Summary

During the final year some progress has been made regarding the CONNECTability. The runtime monitoring and enforcement of security policies are now fully integrated into the architecture such that the required behaviour of eternal interoperability is successfully achieved. Further, the work using the SM@RT tool has demonstrated how the Enabler architecture itself can be managed and changed at runtime to cope with changing demands of its services.

However, limited progress has been made regarding closing the adaptation loop within the architecture itself. Instead we provided illustrative work where adaptation of CONNECTors is achieved using Models@runtime approach. Here where changes are made to the model these are reflected in the running interoperability solution.

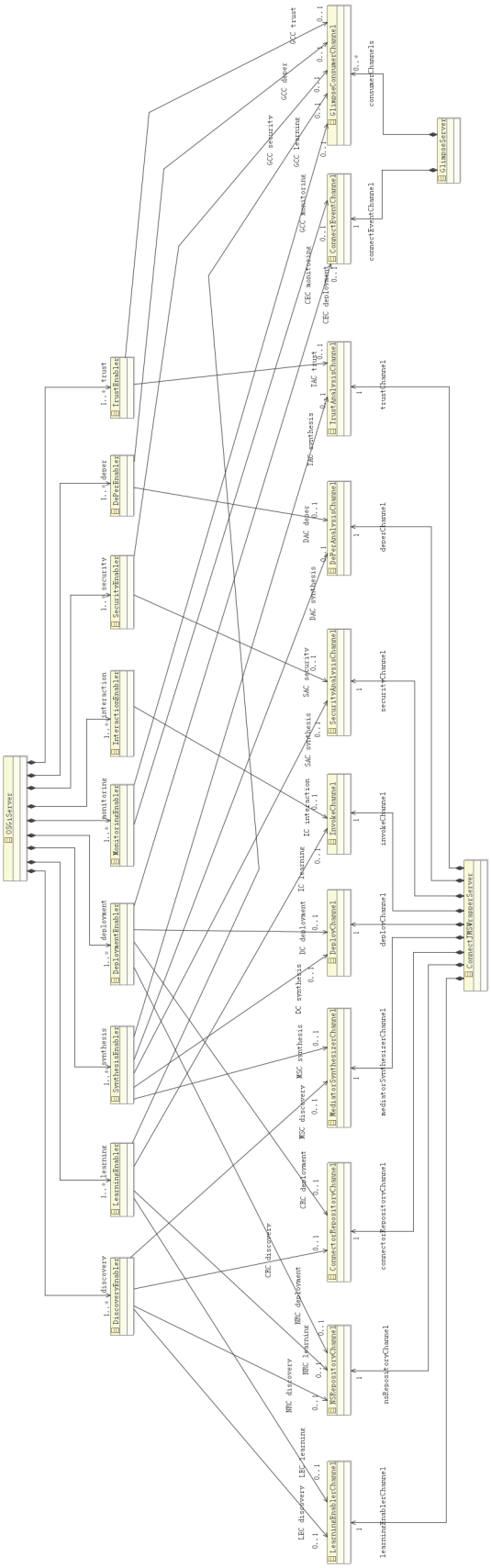


Figure 4.5: Meta-Model of the CONNECT Enabler Architecture

5 CONNECT Ontology Guidelines

5.1 Introduction

Ontologies have been recognised as playing a vital role in overcoming the various challenges found in achieving interoperability in distributed systems [6]. Ontologies provide a means for identifying the key concepts in a domain, and the relationships that hold between them, providing the "common language" which can unite diverse systems that are otherwise unable to interoperate on a technical level. The CONNECT project in particular shows the ubiquity of ontologies in each of the techniques used as part of the overall approach for dynamic interoperability.

In order to achieve interoperability, CONNECT relies on the existence of pre-defined *domain ontologies* that describe all the relevant concepts. However, it is not yet common practice to give fully annotated descriptions of systems using ontologies, although there is movement in this direction with the introduction of SAWSDL[1] [23]. Consequently, it remains in large part an open question as to what form such ontologies should take, and what the best practices are. An analogy can be drawn with programming: while standards such as OWL[2] provide a precise and general means of expression, just as a programming language does, it remains necessary to guide the programmer with well-known solutions, such as design patterns [19]. For example, a domain ontology may be defined at an inappropriate level of abstraction, or from a perspective which is incompatible with our purpose of interoperability, which needs concepts relevant to services, their processes and functionality, rather than, for instance, concepts related to business stakeholders or legal issues.

In this work we examine the requirements placed on the domain ontology by the approaches used in CONNECT and hence elaborate a series of guidelines to be followed when creating, or refining, an ontology that facilitates interoperability of applications working in the domain. These guidelines are used in conjunction with our *ontology language* (i.e. upper ontology) that defines domain-independent concepts and relations pertaining to interoperability concerns, such as the concept "Operation". These general concepts provide additional structure for the domain-specific ontology, similar to the way in which OWL-S[3] provides general concepts for describing services.

Overall the approach described herein sets out the "best practice" for defining domain ontologies for use in CONNECT. Ontologies that fail to adhere to the guidelines remain compatible, in a narrow technical sense, since the correctness of the techniques implemented in the project does not depend on the structure of the ontology. However, an unrefined ontology may make annotation of service descriptions prohibitively difficult, and may prevent matches between services from being found.

5.2 Motivation (CONNECT-specific)

It has been known for some time that certain enablers within the CONNECT architecture make quite specific assumptions about what information is present in the domain ontology, and that it would be fruitful to collate these assumptions into a single document. The issue was subsequently highlighted at the SAC meeting in December 2011:

- Roy Campbell: "Are ontologies totally satisfactory or did they need extending?"
- Brian Randell: "To what extent were there pre-existing ontologies?"

It was also highlighted in the review of March 2012:

- "Most of the complexity of dealing with heterogeneity in large-scale distributed environments has been dumped on the domain ontology. In order to facilitate the future take-up of CONNECT results, the project should come up with a concise description of the role, structure, contents, management, evolution, responsibilities etc. of the domain ontology in the CONNECT framework. This description could be in the form of a "How-to-Guide"."

[1] http://www.w3.org/TR/sawsdl/
[2] http://www.w3.org/TR/owl-features/
[3] http://www.w3.org/Submission/OWL-S/

5.3 CONNECT Overview

The CONNECT project aims to enable eternal interoperability through the provision of dynamically-generated *emergent middleware*. In this approach, a component called a *mediator* is synthesised to overcome incompatibilities between a pair of networked systems (NSs) discovered at runtime such as the syntactic names of operations and the order in which they are applied. Mediators are synthesised at runtime by analysing semantic descriptions of the interface and behaviour (protocol) of each NS. In addition, the approach considers the non-functional aspects of the NSs and the eventual connected system.

The techniques used in the project are implemented in *enablers* that are combined in the CONNECT architecture. The enablers referenced in this document are:

- **The discovery enabler** exhibits a plugin architecture that enables it to use various dynamic discovery protocols (such as WS-Discovery and UPNP) to populate a repository of descriptions of services available on a network. These descriptions follow the form of the networked system model given above. Upon discovery of a new service, this enabler also find pairs of services that have compatible affordances (normally one provided and one required).

- **The synthesis enabler** is invoked (by discovery) when a pair of services are found with compatible affordances. The synthesis enabler analyses the behaviour of the pair of systems alongside a specific goal for the pair and, if necessary, generates a mediator. This mediator is composed with the services (creating a *connected system*), enabling the two services to interact in such a way as to achieve the goal.

The CONNECT project conjectures the use of pre-existing ontologies that describe domain-specific concepts to support the semantic annotation of, and reasoning about, networked systems (NSs). We call these *domain ontologies*. However, it is not certain that every such domain ontology will describe concepts in a manner which addresses the interoperability concerns of CONNECT. To overcome this, we propose that domain ontologies be refined (or created afresh) with the help of the guidelines set out herein and our ontology language. This refinement process should take place offline by a person whom we call the *ontology refiner* (i.e. domain expert), and the refined ontologies (that we call CONNECT *ontologies*) should be made available on the world-wide web. In a subsequent step, the creator of an NS uses the appropriate CONNECT ontology to annotate the description of the NS.

5.4 Networked system model

In order to fully explain our ontology guidelines, we first recall the Networked System Model, which determines the format of NS descriptions used throughout the project.

A description of a networked system is divided into several parts. Figure 5.1 shows an example networked system description for an NS called “Weather Station”, comprising three parts:

- the *affordance* (labelled “functional semantics” in the figure) $a = \langle t, c, i, o \rangle$, which describes the high-level functionality or capability of the NS by referring to three domain concepts where c is the functionality, i is the high-level input, o is the high-level output, and t indicates whether the functionality is provided (by the NS to others) or required (from others by the NS);

- the interface, which lists the signatures of all the NS’s operations $\langle a, I_a, O_a \rangle$, where a is the operation concept[4], I_a is the input parameter concept, and O_a is the output parameter concept; and

- the behaviour, which describes how operations of the interface are co-ordinated in correct use of the NS.

In addition to the evident ontology references in the affordance and interface, an NS should specify its non-functional properties in terms of domain concepts, according to the CONNECT Property MetaModel (CPMM) [12].

[4] Overbar $\bar{a}$ indicates a provided (output) operation, while its absence indicates a required (input) operation.

The affordance part of the description enables the discovery enabler to identify pairs of NSs that have potentially matching functionality without having to perform time-consuming behavioural matching for all possible pairings. The semantically-annotated interface, in conjunction with the behaviour, ensures that the synthesis enabler can find matching operations (or sequences of operations) between two NSs, so as to enable interaction. Finally, the semantically-annotated non-functional description enables pairs of NSs with compatible non-functional properties to be identified.

5.5 Ontology language

Annotating an NS description as above, when presented with an unrefined domain ontology, possibly comprising hundreds or thousands of concepts (Cyc, for instance, comprises 47,000 concepts), is a difficult task for the NS creator. Which concepts, for example, are sensible annotations for interface operations? Our ontology language aims to reduce the effort involved by introducing a minimum level of structure for CONNECT ontologies.

The ontology language introduces several generic core concepts:

- `Affordance`: domain concepts which can be considered as representing high-level functionalities should be marked as subtypes of Affordance.
- `Operation`: domain concepts which can be considered as representing interface operations should be marked as subtypes of Operation.
- `Parameter`: domain concepts which can be considered as representing data parameters should be marked as subtypes of Parameter.
- `NFEventType`: domain concepts which can be considered as representing CPMM event types should be marked as subtypes of NFEventType.
- `NFProperty`: domain concepts which can be considered as representing non-functional properties should be marked as subtypes of NFProperty.
- `NFMetrics`: domain concepts which can be considered as representing CPMM metrics should be marked as subtypes of NFMetrics.
- `NFMetricsTemplate`: domain concepts which can be considered as representing CPMM metrics templates should be marked as subtypes of NFMetricsTemplate.

The language also introduces the generic `partOf` relation to express part-whole relationships. See Section 5.7 for its definition. Fixed relationships between the non-functional concepts are given in the CPMM [12].

In the following we describe the guidelines that are to be followed by the ontology refiner when creating a CONNECT ontology. OWL is assumed as the base ontology language, but this is not a necessary assumption.

5.6 Guidelines

5.6.1 Affordance

The affordance describes the high-level functionality of the NS, and gives rise to the first guideline:

- The functionality concept c in every affordance $a = \langle t, c, i, o \rangle$ must inherit from the class `Affordance`.

  $$\forall a = \langle t, c, i, o \rangle : c \sqsubseteq \texttt{Affordance}$$

  In other words, all the high-level functionalities relevant to the domain must be given as subclasses of `Affordance`. Strictly, the requirement can be relaxed such that c should exist in the ontology, but it may not be specified as a kind of affordance. If it is explicitly stated, this facilitates the training phase of affordance learning [4] used by the discovery enabler by making it easier to see *a priori* which concepts are affordances.

- Two compatible affordances must be in a subtype relation with one another. This is necessary in order for the discovery enabler to find matching pairs of NSs.

The next guidelines relate to the way in which the discovery enabler determines which NSs are relevant to the achievement of the user goal. In particular, the LTL goal formula refers to operations that are necessary for the satisfaction of the goal. In order to find the NSs which can be used to satisfy the goal, the discovery enabler uses the ontology to determine which affordance the operations are related to, and then searches in the repository for NSs with that affordance.

- Every operation op used in the protocol of an affordance must have a `partOf` relation to that affordance or one of the affordance's supertypes.

  $$\forall a = \langle t, c, i, o \rangle : \forall op \in \Sigma_a : isPartOf(op, c)$$

  This requirement in particular allows the discovery enabler to find the affordance associated with the operations referenced in an LTL goal formula.

5.6.2 Operations and data

The interface description requires that each operation be annotated with a concept. In addition the data parameters of each operation should be annotated with one or more concepts. In the case of SAWSDL-based descriptions, the leaf nodes in the tree of XML Schema types defining a parameter must each have an annotation. This leads to the guidelines:

- Every operation is annotated with a concept that inherits from the class `Operation`.

  $$\forall op : op \sqsubseteq \texttt{Operation}$$

  This rule can be relaxed, but adherence to it eases the job of the developer of a new NS who must search through the potentially large ontology to find the right concept with which to annotate the interface.

- Every data parameter is annotated with a concept that inherits from the class `Parameter`.

  $$\forall op : op \sqsubseteq \texttt{Parameter}$$

  This may be relaxed with a similar caveat.

The following guidelines arise from the synthesis enabler that uses the relationships between concepts in the ontology to determine which operations between a pair of NSs are equivalent. This is achieved by finding *mappings* as follows.

- An input operation $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ is satisfied by or *maps to* an output operation $\overline{\beta} = \langle \overline{b}, I_b, O_b \rangle \in \mathcal{I}_2$, denoted $\alpha \overset{1-1}{\longmapsto} \overline{\beta}$, iff:

  1. $b \sqsubseteq a$
  2. $I_a \sqsubseteq I_b$
  3. $I_a \sqcup O_b \sqsubseteq O_a$

  The relationships between data parameters are also used by the learning enabler.

- An input operation $\alpha = \langle a, I_a, O_a \rangle \in \mathcal{I}_1$ maps to a sequence of output operations $\langle \overline{\beta_i} = \langle \overline{b_i}, I_{b_i}, O_{b_i} \rangle_{i=1..n} \in \mathcal{I}_2 \rangle$, denoted $\alpha \overset{1-n}{\longmapsto} \langle \overline{\beta_1}, ..., \overline{\beta_n} \rangle$, iff:

  1. $\bigsqcup_{i=1}^{n} b_i \sqsubseteq a$, i.e., a subsumes $unionOf(b_1, ..., b_n)$. Note that this statement is checked and does not need to be explicitly specified by the designer.

2. $I_a \sqsubseteq I_{b_1}$
3. $I_a \bigsqcup \left(\bigsqcup_{j=1}^{i-1} O_{b_j} \right) \sqsubseteq I_{b_i}$
4. $I_a \bigsqcup \left(\bigsqcup_{j=1}^{n} O_{b_j} \right) \sqsubseteq O_a$

- A sequence of input operations $\langle \alpha_i = \langle a_i, I_{a_i}, O_{a_i} \rangle \in \mathcal{I}_1 \rangle_{i=1..m}$ maps to a sequence of output operations $\left\langle \overline{\beta_j} = \left\langle \overline{b_j}, I_{b_j}, O_{b_j} \right\rangle_{j=1..n} \in \mathcal{I}_2 \right\rangle$, denoted $\langle \alpha_1, ..., \alpha_l, ..., \alpha_m \rangle \overset{m-n}{\longmapsto} \left\langle \overline{\beta_1}, ..., \overline{\beta_n} \right\rangle$, iff:
  1. $\bigsqcup_{j=1}^{n} b_j \sqsubseteq \bigsqcup_{i=1}^{m} a_i$
  2. $\bigsqcup_{i=1}^{l} I_{a_i} \sqsubseteq I_{b_1}$
  3. $\left(\bigsqcup_{j=1}^{l} I_{a_j} \right) \bigsqcup \left(\bigsqcup_{h=1}^{i-1} O_{b_h} \right) \sqsubseteq I_{b_i}$
  4. $\forall h \in [1, l[,\ O_{a_h} = \emptyset$
  5. $\forall h \in [l, m], \left(\bigsqcup_{i=1}^{h} I_{a_i} \right) \bigsqcup \left(\bigsqcup_{k=1}^{n} O_{b_k} \right) \sqsubseteq O_{a_h}$

5.6.3 Non-functional concepts

In order to enable matching between two concepts p and q (inheriting from `NFProperty`, `NFMetrics` or `NFMetricsTemplate`), the following condition should be met:

$$\exists z : p \sqsubseteq z \wedge q \sqsubseteq z \wedge z \neq \texttt{NFProperty} \wedge z \neq \texttt{NFMetrics} \wedge z \neq \texttt{NFMetricsTemplate}$$

In other words, a match between non-functional properties is possible where the concepts share a common (specific) ancestor.

5.7 Definition of `partOf`

OWL has no native (reasoning) support for `partOf` [1]. For this reason we make our definition explicit. Indeed it coincides with *ground mereology* [34] and W3C recommendations [1].

Definition 1 (`partOf`)**.** *`partOf` is a reflexive, transitive and antisymmetric (binary) relation between classes. The notation pPw (equivalently $P(p, w)$) means the part p is a part of the whole w.*

$$aPa \tag{5.1}$$
$$aPb \wedge bPc \longrightarrow aPc \tag{5.2}$$
$$aPb \wedge bPa \longleftrightarrow a = b \tag{5.3}$$

For convenience we denote the inverse relation `hasPart` as wHp. `hasPart` is inherited, while `partOf` is not[5]*.*

$$aPb \longleftrightarrow bHa \tag{5.4}$$
$$b \sqsubseteq a \wedge aHc \longrightarrow bHc \tag{5.5}$$

[5]Since we speaking about classes. Consider the "eye" class which is a part of the "body" class. In other words, all instances of eyes are parts of instances of bodies, and all bodies have eyes. It is not safe to say the "dog eye" class is a part of the "body" class (through inheritance) since there are some bodies which do not have dog eyes, but other kinds of eye.

It is also useful to define the composition (sum) of parts:

Definition 2 (Sum of parts)**.** *The notation $a \oplus b$ defines a third class which contains all the parts of a and b (by transitivity of P), and no part which is not in a or b.*

$$aP(a \oplus b) \tag{5.6}$$
$$bP(a \oplus b) \tag{5.7}$$
$$a \oplus b = b \oplus a \tag{5.8}$$
$$zP(a \oplus b) \longrightarrow zPa \vee zPb \tag{5.9}$$

Given these definitions we can specify the condition under which a whole can be substituted by a sum of parts (and vice versa):

Definition 3 (Substitution)**.** *A whole w can be substituted with a sum of parts p_i only when the sum contains at least all the parts of w (and possibly more).*

$$wP \bigoplus_i p_i \tag{5.10}$$

5.8 Definition of Sequences

OWL has no built-in construct for sequences/ordering. Sequences are important to describe causally-related events (e.g., sequence of musical events (chords) [] or action chain of a plan [8]) as well as liked structures (e.g., protein sequences [17] or strings [25]). To rely on Drummond *et al.* definition [17], which is in line with W3C best practices[6]. It is based on the definition of two functions `hasNext` and `hasContent` , and a transitive relation `isFollowedBy`. In addition, `EmptySequence` specifies the element that has neither content nor following element. where EmptySequence $\doteq \exists$hasContent.$A \sqcap \exists$hasNext.$(\exists$hasContent.$B \sqcap$ hasNext.$(\exists$hasContent.$C \sqcap \exists$hasNext.EmptySequence$))$

For example, The concept S defined as the sequence of concepts A, B, and C is defined as follows: $S \doteq \exists$hasContent.$A \sqcap \exists$hasNext.$(\exists$hasContent.$B \sqcap$ hasNext.$(\exists$hasContent.$C \sqcap \exists$hasNext.EmptySequence$))$

5.9 Example

To illustrate the ontology refinement process we consider a simple ontology for the weather domain that might be used to annotate the description in Figure 5.1. Figure 5.2 shows the initial ontology, with very little structure to guide the NS creator in finding appropriate concepts for the various parts of the NS description. Without auxiliary documentation, the annotation process is essentially guesswork based on the names of concepts, which are often poor.

We suppose that the ontology refiner takes the bare ontology and uses our ontology language, in the first step, to annotate the domain concepts with the general concepts from our language. This provides the initial structure useful to the NS creator. Figure 5.3 shows, for instance, that “Records” is an affordance concept (an entity that keeps records) rather than an item of data, and that “Station” is a kind of data parameter.

The ontology refiner can go further and elaborate additional information for use by the CONNECT enablers, as in Figure 5.4. In particular, a subsumption relation has been added to indicate that the “GetRainSensor” operation is a kind of “GetSensor”, making them interchangeable under certain conditions. “GetWeatherData” has been marked as equivalent to the union of “GetSensorData”, “GetWeatherStation” and “GetSensor”, meaning that a sequence of the latter three can be performed to achieve “GetWeatherData”. Similarly, “Login” is marked as a union of “CreateProp”, “CreateSession” and “Authenticate”. Finally `partOf` relations are added between operations and affordances, such as between “GetWeatherData”, “Login” and the affordance “Consumer”.

This refined ontology expresses information that was perhaps implicit in the original ontology. The refined ontology aids the NS creator who must accurately annotate NS descriptions, and ensures that matches can be found when subsequently applying the CONNECT approach.

[6] http://www.w3.org/TR/swbp-n-aryRelations/

5.10 Related work

As alluded to above, our ontology language can be regarded as an upper ontology, to which is attached the lower domain-specific part. Several proposals have been made in the past for upper ontologies of varying generality. Cyc[7] and SUMO[8] are the largest general ontologies available today, with around 47,000 and 25,000 concepts respectively. DOLCE[9] is another example, albeit with a linguistic bent. Both OWL-S[10] and our language are in comparison much more specific, being geared to the description of (computational) services. OWL-S is somewhat more heavyweight, in the sense that instantiating it to a particular domain requires describing every service in detail using the concepts of the upper ontology. In contrast, we expect only that the domain concepts are divided into categories like affordance, operation, and so on.

Other work following our approach of defining a general ontology for interoperability, which is then combined with a domain-specific part, includes Gu *et al.* [22], where the authors define concepts for describing context, and Hunter [27], where an upper ontology for multimedia formats is defined.

5.11 Conclusions

In this work we have set out the "best practice" for defining (or refining) domain ontologies for use in the CONNECT project and more generally in software interoperability. Our approach comprises an ontology language and an associated set of guidelines which, if adhered to, will ensure that interoperable pairs of services can be found and that the effort of annotation undertaken by NS creators is not prohibitive. In particular, our language makes clear that it is helpful for the domain ontology to have a perspective relevant to the specification of networked systems, comprising concepts and relations for high-level functionality and lower-level operations.

We have, however, only been concerned with the structure and content of the ontology. There are a series of related questions that should be addressed in future work such as how domain ontologies are best managed and in particular how an evolving ontology impacts the assumptions made by each enabler, and whether such evolution can be managed through adaptation of the connected system.

[7] http://www.cyc.com/opencyc/
[8] http://www.ontologyportal.org/
[9] http://www.loa.istc.cnr.it/DOLCE.html
[10] http://www.w3.org/Submission/OWL-S/

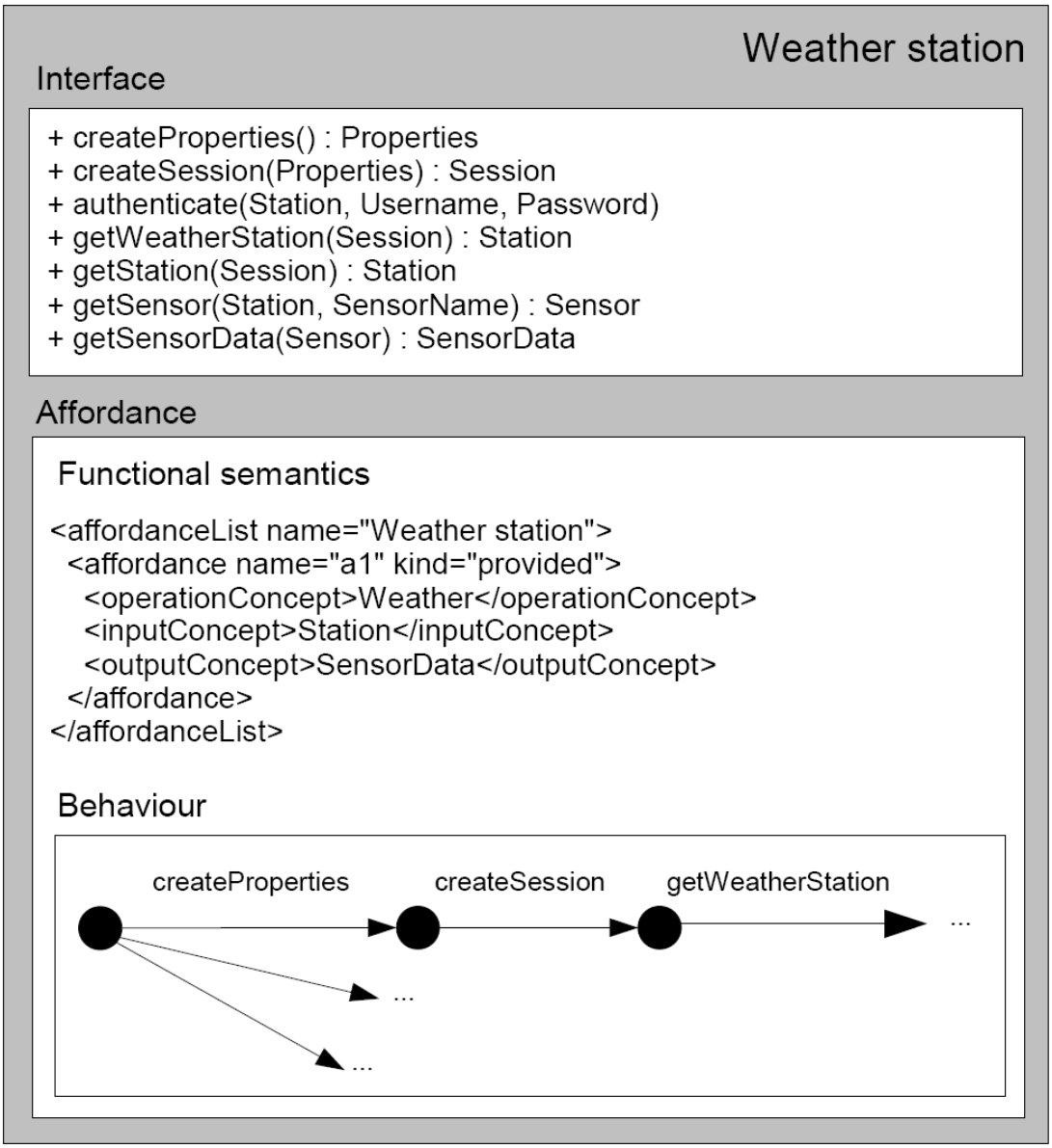


Figure 5.1: Networked system description

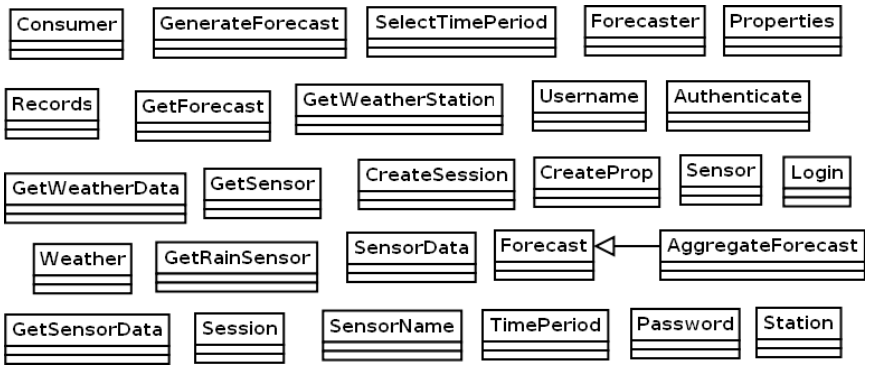


Figure 5.2: Unrefined domain ontology (UML class notation)

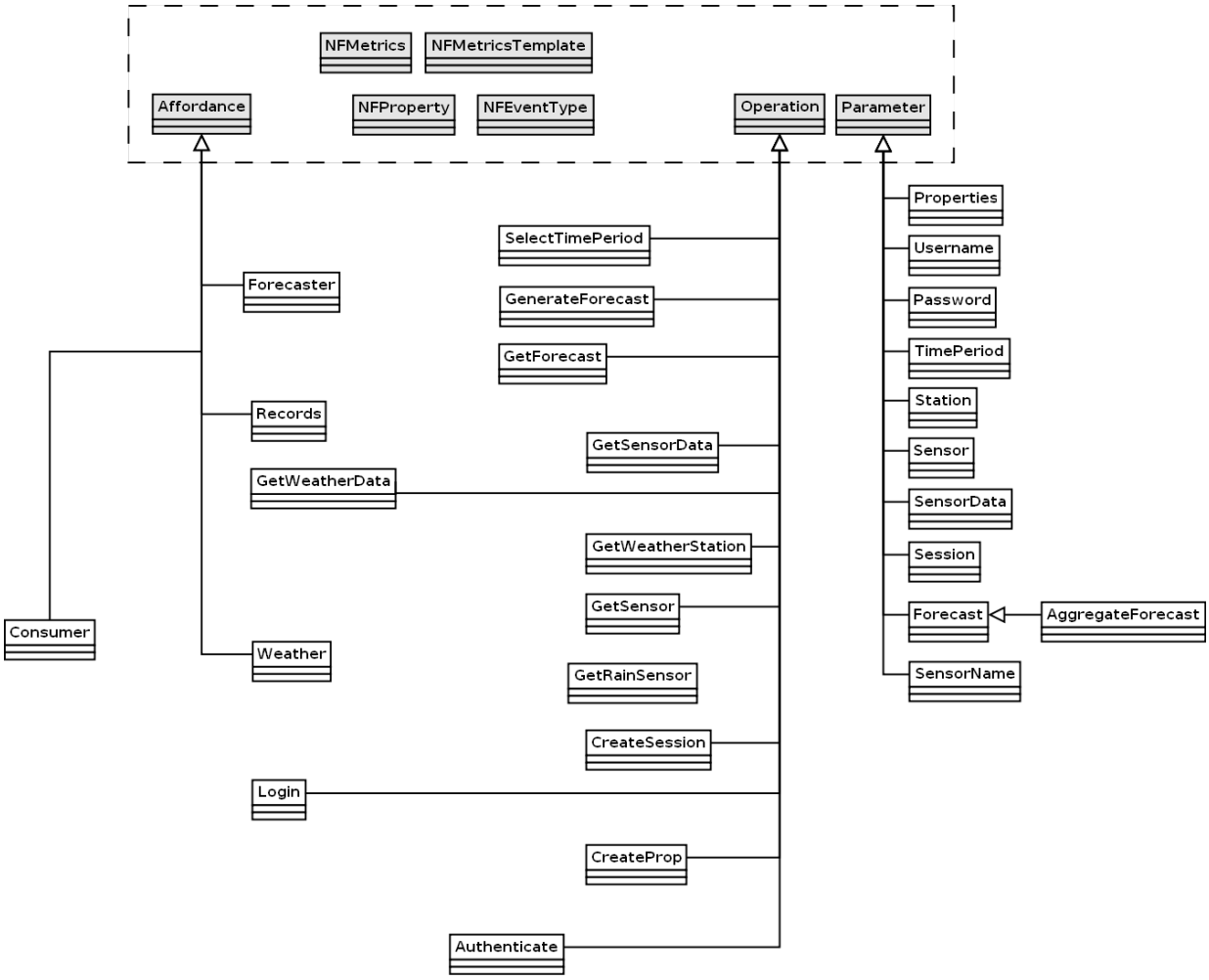


Figure 5.3: Annotated domain ontology

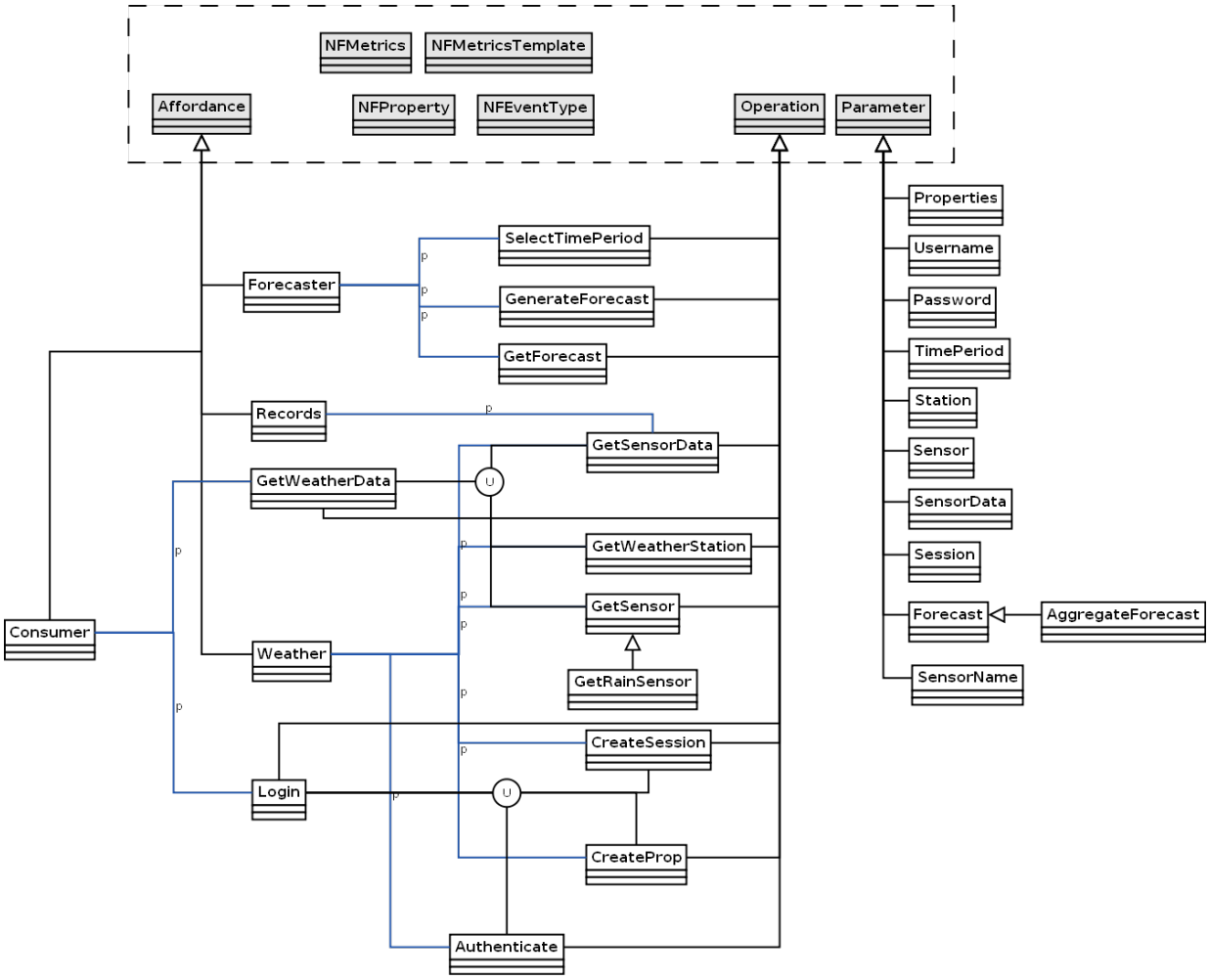


Figure 5.4: Domain ontology fully refined for CONNECT (blue lines marked with "p" indicate `partOf` relations; circled 'u' indicates a union)

6 Synthesis of CONNECTors for Mobile Applications

6.1 Introduction

In this section, we discuss how the architecture of modern mobile platforms and ecosystems creates additional barriers for interoperability. Mobile platforms are becoming increasingly popular, and accounted for an average of 13% of the world's Internet traffic on November 2012 according to the StatCounter website[1] . Individuals are progressively using their mobile devices to access services on the Web and are starting to realize the harmful effects that heterogeneous services with incompatible interfaces can have on their mobile activities. This section focuses on the connection of NSs in the context of mobile apps, thereby considering interoperability across apps as well as the interoperability of apps with Internet-based services. While the synthesis of abstract mediators is not impacted by the specific application domain, the further synthesis and deployment of emergent middleware executing CONNECTors must account for the characteristics of mobile platforms

In Section 6.2, we present the features of modern mobile ecosystems in order to explain in Section 6.3 whether the existing CONNECT architecture is suitable for mobile mediation. Section 6.4 then details how we extended the CONNECT architecture to mobile environments, using mechanisms for inter-application communication that are typical of modern mobile operating systems. Finally, in Section 6.5 we present two applications that we developed using the CONNECT mobile extension: one that provides users with access to different cloud data storage services and another that enables a legacy application to store data on a cloud storage service originally unknown to the application.

6.2 The Ecosystem of Modern Mobile Platforms

The transition from desktop to mobile computing caused a rupture on the architecture of operating systems and applications. Mobile devices are not just smaller desktop computers: they are fundamentally different from desktop computers in the way they access the network, on how they manage third-party applications and how these applications access the users' resources and contents.

Compared to desktops, mobile devices operate in an environment with different stakeholders (e.g., mobile phone operators) and additional requirements (e.g., phones should not crash in the middle of an emergency call because an application crashed). Modern mobile platforms (the mobile operating system and software running locally on the device) and ecosystems (the combination of services supporting the mobile user and his device, and the network infrastructure) are designed with such requirements in mind, which ultimately impacts the way applications are designed and the functionalities they can provide to users. In the following sections, we detail the features that differentiate modern mobile platforms and ecosystems from previous generation desktops.

6.2.1 Application Lifecycle

On desktops, applications are installed from a physical storage media (such as a USB key) or downloaded from the application developer website. Users generally download a binary file but it is also possible to download the application source files and compile them on the desktop. Some operating systems (such as Windows 8 or OSX Lion) provide mechanisms to verify the application trustfulness through cryptographic signatures and others (such as Debian Linux) provide advanced dependency-checking tools to ensure that all the software needed to run an application is installed on the desktop. Since binary authentication is not mandatory, applications can even dynamically generate code and modify their behavior at run-time.

On mobile devices, operating systems emphasize reliability (and more lately privacy) over flexibility. Some services provided by mobile devices, such as phone calls and SMS, must always be available regardless of the state of the applications running on the device. As a result, greater control is enforced on application installation and distribution to reduce the possibility of crashes and of the mobile device

[1] bit.ly/VjAAaW

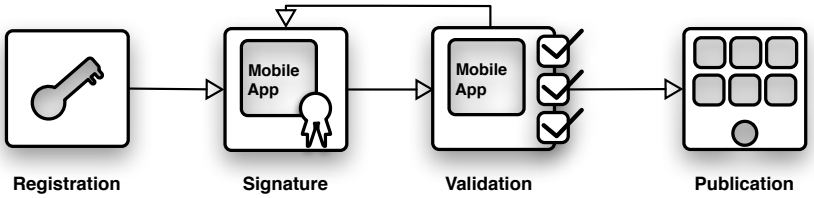


Figure 6.1: The Process of Mobile Application Release

becoming unavailable as a result of application malfunctioning. Modern mobile ecosystems, thus, provide **application marketplaces** that curate and distribute applications to mobile devices. Applications can be tested before being available on the marketplace (iOS) or not (Android), and applications with problems or reported as malicious can be removed from the marketplace and even remotely removed from the devices themselves. As illustrated in Figure 6.1, applications are cryptographically signed by a registered user, and the marketplace acceptance process can take from minutes on Android to days on iOS. Although technically feasible, applications cannot dynamically generate code since (i) it would void the marketplace validation and (ii) it is forbidden by some platforms terms and conditions.

6.2.2 Network Connectivity

Desktop computers are usually connected to the same trusted network for extended periods of time. Whether it is a home or an office computer, the user trusts the network to which the computer is connected. Moreover, the user is either the network administrator or has access to the network administrator to request changes on the network topology to enable execution of new services. Since the user of a desktop computer uses the same network for a long period of time, it is feasible to deploy a new server on the network as a result of a user requirement because the new server will also be used for an extended period of time.

In the mobile ecosystem, mobile devices connect frequently to untrusted wireless networks and to cellular data networks. Whether the user is at a coffee shop, a park or a hotel, the user may not trust the wireless network to which its mobile device is connected. Such connections are short-lived and may happen a single time or only a few times over the lifetime of the device. Given the ephemeral nature of such network connections, changes to the network topology as a result of a user requirement are unfeasible. When mobile devices use cell phone connections for data transfer, the network infrastructure is managed by a cell phone operator and shared with thousands of users. Changes to the network topology as a result of a service or a user requirement are also unfeasible in this scenario.

6.2.3 Inter-Process Communication

Desktop operating systems provide a number of mechanisms for inter-process communication (IPC), which enables data exchange among processes from different applications. For the particular case of applications running on the same computer, processes can communicate data via signal, semaphore, message queue, pipe, shared memory, message-passing, memory-mapped files, sockets and files.

Modern mobile operating systems isolate processes on **sandboxes**, and thus all IPC mechanisms that require direct data exchange between processes with operating system support (such as signal and semaphore) are not available on mobile operating systems. IPC-specific APIs such as message queue, pipe, shared memory, message-passing and memory-mapped files are also unavailable on modern mobile operating systems. As discussed below, files and sockets are available for mobile applications, but are not suitable for inter-process communication on mobile.

File IPC

Desktop applications generally have full access to the desktop file system. Some particular cases of desktop applications (e.g., java applets) are restricted to sandboxes, and only have access to a part of the file system that is dedicated to the application and isolated from other applications. To reduce interference among applications, modern mobile operating systems restrict each application to its sandbox, and files

from one application are not accessible by another. On iOS, applications can only share data if they exchange cryptographic keys allowing access at compilation time (which is uncommon for applications from different vendors), while on Android an application must get explicit consent from the user at run-time to access data from another application. As a result, although available on modern mobile operating systems, files are not suitable as a general mechanism for mobile IPC.

Socket IPC

Desktop operating systems provide **multi-tasking** capabilities to applications, enabling them to share computing resources such as CPU and memory with other applications running simultaneously and also allowing applications to run on background while other applications have the main user focus. This allows a network socket to remain open on the background while the user is focused on writing a mail message, for instance. If one of the applications running simultaneously is unstable, users can explicitly kill the application. The operating system, however, does not monitor the application state to interrupt application execution.

Full multi-tasking is available on some mobile platforms (Android) but on other platforms multi-tasking is mediated by the operating system (iOS): the operating system is responsible to wake up applications when events of interest to the application occur, and applications have a limited execution time after which they are preempted. Applications thus run on a simulated multi-tasking environment while the operating system is actually handling events that are dispatched to the application on very specific cases. For example, each application can have a single socket running on background. If the application takes too long to process a packet received, the operating system can kill the application. Once again, sockets are available for mobile applications, but are not a reliable mechanism for mobile IPC.

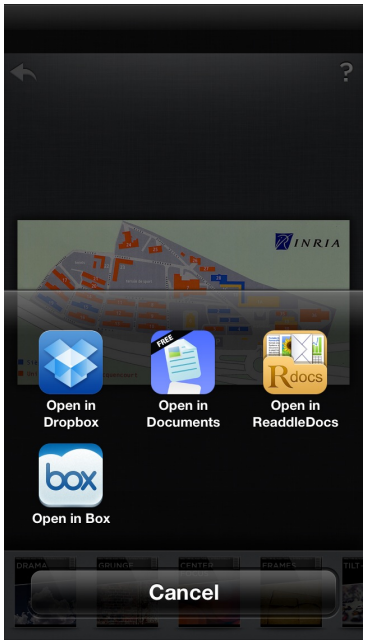
Inter-Application Communication

Instead of traditional IPC mechanisms, mobile platforms provide additional mechanisms for coarse-grained **Inter-Application Communication**. Android provides **intents** that applications can use to declare their ability to handle a certain action. Intents consist of a category and an action, and categories can be used to group related actions. An application, for example, can declare that it is able to handle the action *open* of the category *file*. Some intents are defined by the operating system by default, but applications can also create application-specific intents. Whenever an application wants to send data to another, it asks the operating system the list of applications that declared being able to handle a certain intent, and the list of applications is presented to the user who selects one application from the list. There is no API for an application to discover a list of intents or if a certain application handles a given intent. Both applications must agree beforehand on the syntax and semantics of intents.

Intents are not available on iOS, which provides **URL schemes** for inter-application communication instead. As with Android, mobile applications also register to handle specific URLs (such as `fb://profile` for opening the user profile on the Facebook application) and other applications can ask the operating system to open a given URL. It is also not possible for an application to discover all registered schemes or which application handles a given scheme. The syntax and semantics of URL schemes must be agreed upon beforehand by both the application calling the scheme and the one handling it. Contrarily to Android, iOS chooses at random the application which will handle the scheme: if two applications register for the same scheme, which one the operating system calls is unknown.

Additionally to URL schemes, and since applications on iOS cannot access files outside their sandbox, the operating system provides another mechanism for inter-application communication: registered **file types**. Applications can similarly register with the operating system which file types they are able to handle, and other applications can ask the operating system to open a file with a certain type. Again, in this case, it is not possible for an application to discover which file types are registered or if an application can handle a given file type. Unlike URL schemes, iOS proposes graphically to the user a list of applications able to handle a certain file type whenever multiple applications register for the same file type (see Figure 6.2).

(a) iOS Menu



(b) Android Menu

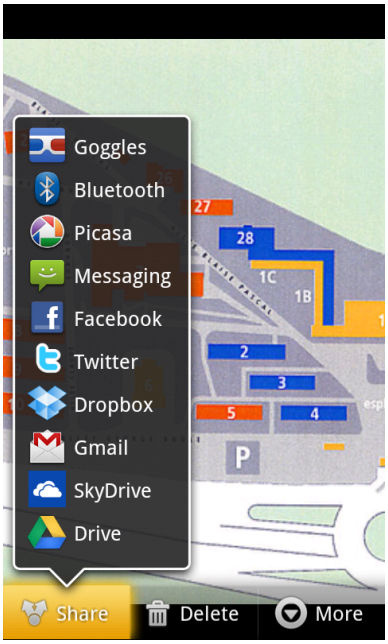


Figure 6.2: Displaying Applications List

6.3 Is CONNECT Suitable for Mediation in Mobile Environments?

In order to enable the interoperability vision of CONNECT on current-generation mobile platforms, two issues must be addressed:

- Deploying the CONNECT Enabler architecture in such a way that it is accessible to mobile systems, in order to collaborate and produce a CONNECTor when required, and
- Enabling CONNECT to deploy CONNECTors on mobile devices, or other systems that are accessible to the CONNECTed mobile systems.

6.3.1 Deployment of CONNECT Enablers

Taking into account the fundamental differences in the architecture of modern mobile operating systems, and the frequent context changes caused by mobility, we cannot assume that the CONNECT Enablers can be deployed (and re-deployed, when switching to different networks) independently of the network topology. Thus, we see three possible solutions for the deployment of CONNECT Enablers in a mobile environment:

- The Enablers are deployed on a single mobile device, providing isolation from the changes in networking conditions. This type of deployment is hard to achieve on current generation mobile devices. One of the reasons is that smartphones lack the necessary resources such as processing-power and battery-life to perform tasks like Active Learning.
- The Enablers are deployed on a specific network, resulting in CONNECT being restricted to certain contexts (e.g., only when connected to a single WiFi network), reducing the overall utility of the approach.
- The Enablers are deployed on the Internet and it is thus assumed that the smartphones can connect to the services at any time over the Internet. But, the deployment on the Internet (or Cloud) poses new technological challenges. One example are discovery mechanisms, which were developed almost exclusively for use on local network environments and are, thus, not easily extendable for use over the Internet.

In response to the above challenges, in Deliverable 1.3, we presented a partial and lightweight CONNECT Enabler implementation, called iBICOOP, which can be deployed on current-generation mobile devices, and allow interaction with the Enablers over the Internet. Further in this chapter, we focus on the challenges of design and deployment of CONNECTors on mobile systems.

6.3.2 Deployment of CONNECTors

Next to make available Enablers to mobile platforms, the second issue is to deploy the CONNECTor in such a way, that the CONNECTed mobile NSs can seamlessly interact through it. We have already seen in Section 6.2 that modern mobile platforms and the overall ecosystem present major differences compared to the classical desktop Personal Computers. Thus, the CONNECTor architecture must be adapted for the mobile environment to address three main challenges: (i) Mobile Deployment, (ii) Mobile Connectivity, and (iii) Mobile Privacy.

Mobile Deployment. As described in Section 6.2, the deployment of applications, and thus, CONNECTors on mobile platforms, faces more restrictions than on desktop computers. We know that mobile devices regularly connect to untrusted networks for short periods of time. This is why mobile CONNECTors should be deployed either locally on the mobile devices requiring mediation, or on a device which is likely to share the same network context with the CONNECTed systems for long periods of time. However, the dynamic deployment of code on mobile platforms is very restrictive. This is mainly due to the way third-party applications are managed. Considering the case where a CONNECTor is deployed beforehand on the same mobile device as the CONNECTed NS, the mobile platform must allow the background execution of the CONNECTor. As we have seen in the previous section, mobile platforms are also restrictive with respect to background task execution, and background access to resources such as network sockets.

Mobile Connectivity. Assuming that a CONNECTor can be deployed and ran as a background task on a mobile device, there is also a requirement of inter-application communication. CONNECT is designed to enable interoperability by using any means of inter process communication available like, for example, shared memory. On modern mobile platforms, running processes are sandboxed, providing isolation between separate applications collocated on a device. We have seen in Section 6.2 that classical IPC methods like pipes and shared memory are not available. Instead, mobile operating systems provide additional mechanisms and abstractions for coarse-grained inter-application communication. We believe that these additional communication paradigms are interesting to exploit for the purpose of CONNECT interoperability.

Mobile Privacy. Modern mobile devices have become a store for sensible personal data. With smartphones being used as a means of payment, and for storing personal information such as contacts, there is definite need to increase the level of privacy on such devices. In this context, executing a CONNECTor on the same device as the CONNECTed service, enables the containment of sensible data on the device. Furthermore, even when running on the same device, users may be unwilling to provide credentials to a CONNECTor for accessing an external service (e.g., Facebook). Taking into consideration the mobile inter-application communication, a CONNECTor may CONNECT two apps deployed on the same device using intents (on Android) or url schemes (on iOS), in which case the CONNECTor does not need access to credentials; For instance, the CONNECTor may use the official Facebook app to communicate with the remote Facebook service, as opposed to connecting directly to the service over the network, where the app enforces the required security over private data.

6.4 Revisiting the CONNECTor Architecture for Mobile Deployment

Following the previous discussion, we first introduce the two types of CONNECTors envisioned for mobile platforms and then show how the current CONNECT architecture is extended with these types of CONNECTors.

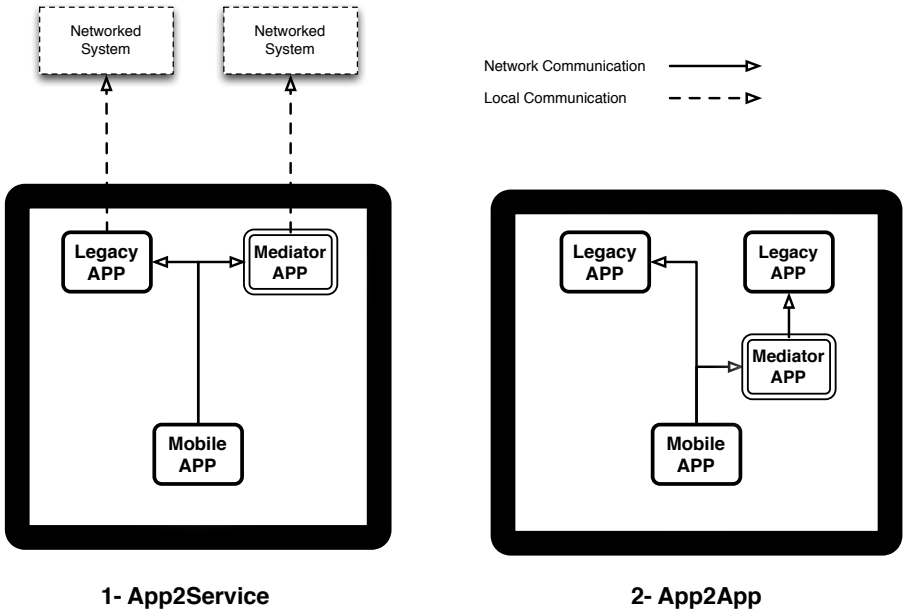


Figure 6.3: Mobile CONNECTors

6.4.1 Mobile CONNECTors

We consider two CONNECTor types (see Figure 6.3) for mobile apps, namely: App2Service and App2App.

- App2Service CONNECTion (see Figure 6.3(1)) concerns mobile applications that were initially designed to rely on services from co-located applications. In this case, the CONNECTor provides the exact same capability as the legacy application but connects directly to an alternative Networked System.
- App2App CONNECTion is illustrated in Figure 6.3(2). Similarly to the CONNECT Networked Systems, co-located applications may provide and require interfaces that are syntactically incompatible but semantically compatible. Hence, we can co-locate with those applications an App2App CONNECTor that allows them tp interoperate.

6.4.2 Mobile CONNECTor Architecture

Consider CONNECTor architecture for the desktop version that is mainly composed of the following elements:

- A *Network Parsers* receive and parse middleware-specific messages. Parsers then produce abstract messages providing a uniform representation of networked system actions that can be manipulated and understood by the other elements of the CONNECTor architecture. An action abstracts middleware primitives (e.g., read, write, send, receive, invoke etc...) and is mainly defined by a source, a destination, a name and an input and an output data.
- A *Network Composers* perform the reverse role of the parser. They compose middleware-specific network messages. Composers specifically receive abstract messages and produce middleware messages to be sent over the network.
- The *Automaton Engine* enables running synthesized CONNECTors. Its role is to interpret mediators specified as automata, which state how to coordinate the protocols run by CONNECTed systems through adequate message translation and buffering.

The mobile version of the CONNECTor architecture then adds the following elements:

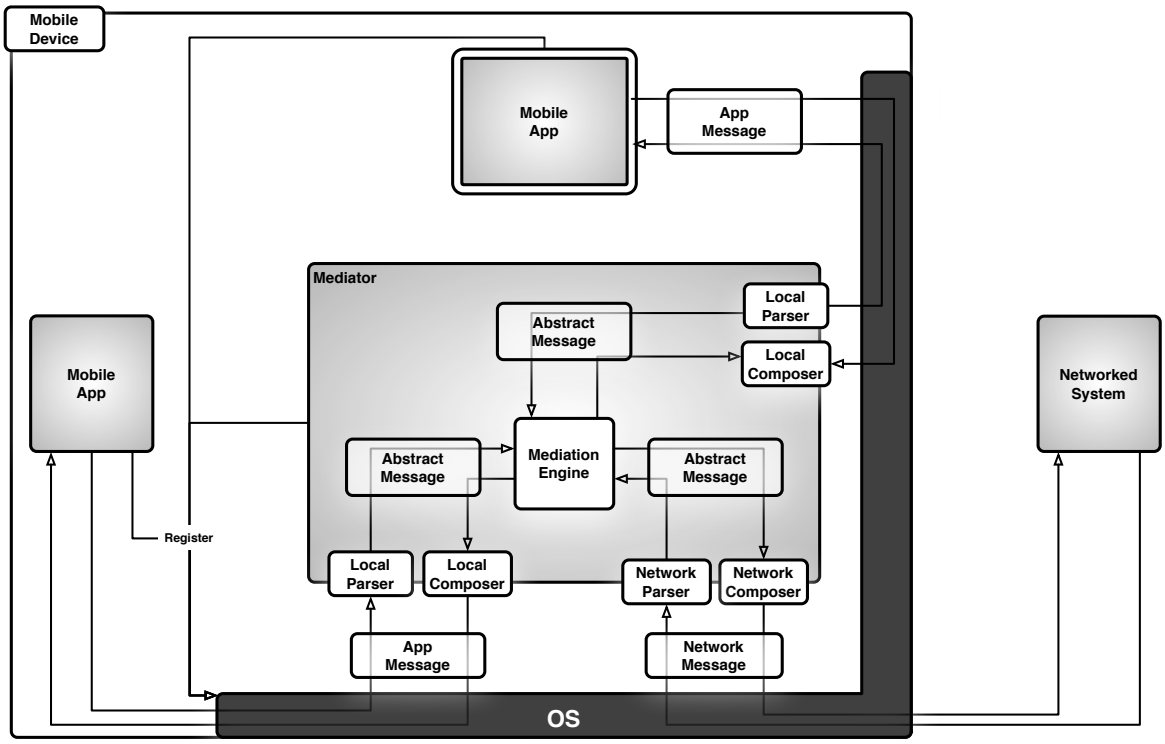


Figure 6.4: Mobile CONNECTor Architecture

- *Local Parser* and a *Local Composer* are defined to handle specific application messages (see Figure 6.4). As discussed in the previous section, the content of the application messages depends on the mobile platform. On Android, the message content is encoded into an Object representation, called intent, whereas on iOS the message content is encoded as URL or a File. Table 6.1 displays how the local parser and composer are managed on Android and iOS. As illustrated, the Android message specification is very close to the CONNECT abstract message representation, thus parsing and composing them is straightforward. On the other hand, the specification of iOS messages is very generic and requires the developer to define a proprietary protocol to encode and decode his application data into a File or a URL.

Abstract message	App message		
	iOS		Android
	URL	FIle	
Destination	URL Name	File Type	Intent category name
Source	URL Name	File Type	Intent parent
Action Name	URL content	File content	Intent action name
Action IO Data	URL content	File content	Intent data / Intent extra data

Table 6.1: Mapping App Message to Abstract Message

- *Registration* is another key aspect of mobile CONNECTion. It defines the way a legacy application may find, and communicate via, co-located CONNECTors. To do so, at design-time, a CONNECTor has to define the application capabilities to which it allows CONNECTing. For instance, if a service requires using a cloud storage capability, then the corresponding CONNECTor has to declare to the operating system its capability to handle messages related to cloud storage. In more detail:

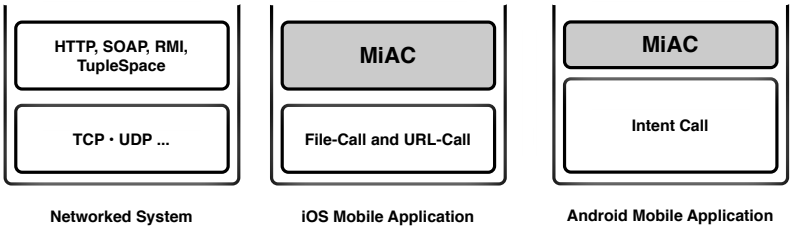


Figure 6.5: MiAC Implementation

– For the iOS platform, the CONNECTor can specify a URL Scheme defined in the Application Manifest, which represents the cloud storage capability, as follows:

```
01) <key>CFBundleURLTypes</key>
02) <array>
03)   <dict>
04)     <key>CFBundleTypeRole</key>
05)     <string>Editor</string>
06)     <key>CFBundleURLName</key>
07)     <string>com.ambientic.cloudstorage</string>
08)     <key>CFBundleURLSchemes</key>
09)     <array>
10)       <string>cloud</string>
11)     </array>
12)   </dict>
13) </array>
```

This description enables the CONNECTor to be invoked whenever an application triggers a call to a URL with the scheme *cloud* (line 10).

– For the Android platform, the CONNECTor can define inside its Application Manifest a filter on an Intent category that represents the cloud storage capability, as follows:

```
01) <intent-filter>
02)   <category android:name="com.ambientic.cloudStorage" />
03) </intent-filter>
```

This description enables the CONNECTor to be called whenever a collocated application instantiates an Intent with the category set to *com.ambientic.cloudStorage*.

6.4.3 MiAC: Mobile inter-Application Communication Middleware

The local parser and composer of a mobile CONNECTor must handle application data following a custom structure. Hence, it is hard, if not impossible, to automatically understand, parse and compose application data. The complexity of this task is also related to the mobile platform.

The Android Intent data are well-structured since developers follow a consistent pattern of defining Intent attributes (i.e., abstract action name becomes an *Intent Action Attribute*, and their in/out data as a dictionary with the Intent *Extra-data Attribute*).

However, nothing prevents Android developers to encode an action's data using a custom URL or into a file. Unfortunately, this is the only way for iOS developers to send their data from an application to another. Another limitation concern the way an application can answer to a given request. For Android, when an intent is invoked, the called application can natively answer to the caller (the Intent parent). On iOS, developers have to manually encode the caller address (e.g, URL scheme) into the request in order to enable the called application to identify and return results to its caller.

As illustrated in Figure 6.5, iOS offer a basic and generic inter-Application Communication mechanism, while Android offers a more sophisticated implementation. Both can be intuitively compared to the socket communication mechanism used by middleware to implement a more sophisticated behavior.

We introduce MiAC, a new Mobile inter-Application Communication Middleware to harmonize mobile platforms and enable a message-based and RPC-based communication with a unique way of encoding

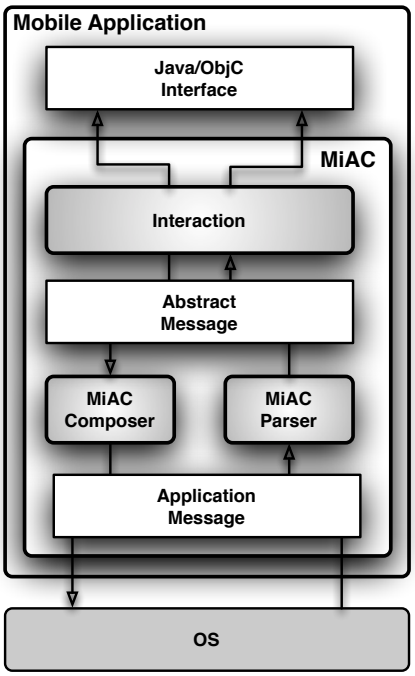


Figure 6.6: MiAC Architecture

application messages. MiAC overcomes the lack of a proper middleware for inter-application communication interfaces on both Android and iOS. The implementation of MiAC for Android is a lightweight layer that maps the MiAC API to the Android Intent API, while on iOS it is more complex as it requires building a full RPC communication middleware.

The middleware architecture illustrated in Figure 6.6 is made up of the three following elements:

- The *Interaction Component* is the middleware entry point. It enables developers to define required and provided actions in the form of Java or ObjectiveC interfaces and takes care of mapping them into abstract messages.

- The MiAC parser and composer are responsible for interacting with the operating system by encoding/decoding abstract messages into/from application messages according to the specification illustrated in Table 6.2.

Abstract message	Destination	Source	Action name	Action input	Action output
Intent	Category	Intent Parent	Action	Extra-data	Extra-data
URL	URL scheme	URL domain	URL path	URL query	URL fragment
File	File type	JSON encoding			

Table 6.2: MiAC Mapping of an Abstract Message to an Application Message

The MiAC parser and composer are included in the mobile CONNECTor architecture, automatically enabling all mobile applications that comply with the specification given in Table 6.2 to be easily CONNECTed. However, any other custom specification will require a learning/implementation phase to customize/design their corresponding parser and composer.

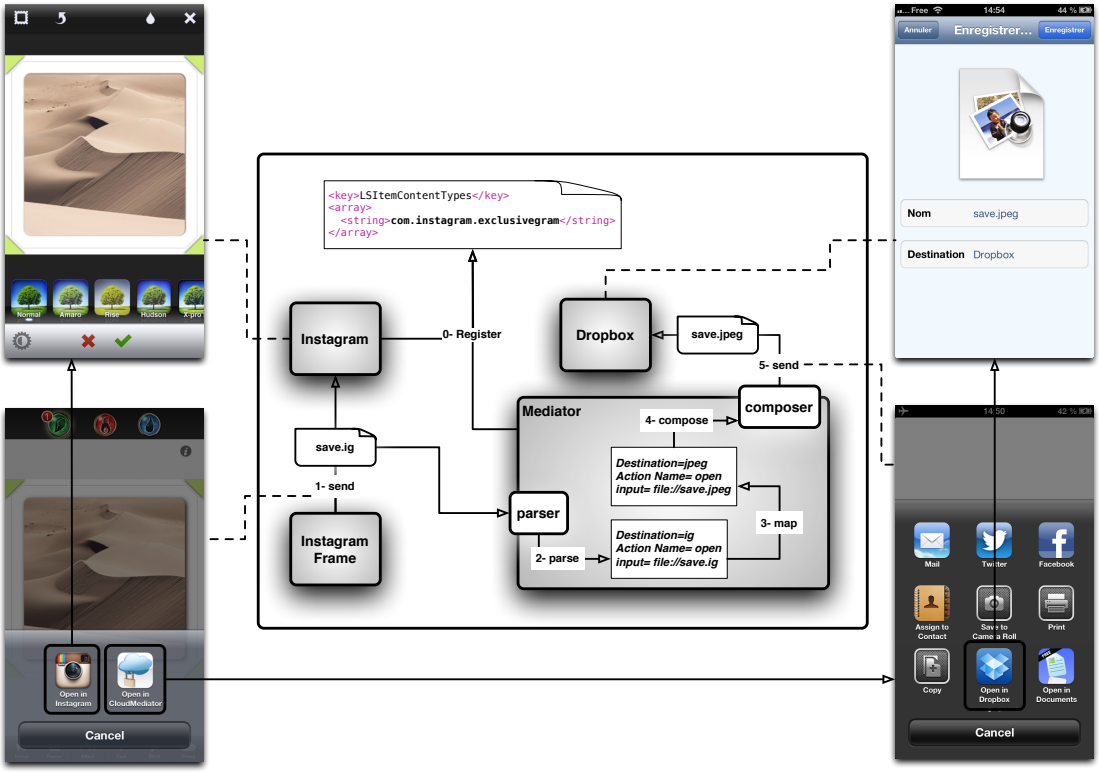


Figure 6.7: Instagram CONNECTor

6.5 Demonstrator: Cloud Application Storage

We carried out two experiments with mobile CONNECTors on the iOS platform. One involves Instagram legacy applications that use a custom application data format and the other is an in-house cloud application based on MiAC.

Instagram Legacy Applications

The first experiment involves the Instagram and Instagram Frame applications, both of which have been developed by Instagram. The first application is used to capture, adjust and share pictures. The second application allows pictures to be edited by adding picture frames. Users can also share their framed photos from the Instagram Frame application to the Cloud via the local Instagram application. In order to extend the sharing capability of the Instagram Frame application, we developed a CONNECTor (called *CloudMediator*) that handles the same Instagram data types (i.e., *com.instagram.exclusivegram*) and enables pictures to be shared with many other cloud applications installed on the mobile device (see Figure 6.7).

When *CloudMediator* is deployed on the device, the operating system allows the Instagram application to find the CONNECTor and share images with CONNECTed Cloud services. The user can select *CloudMediator* to share his edited pictures, which leads the *Instagram Frame* application to send files over a proprietary extension "*ig*" (Figure 6.7 step 1). *CloudMediator* receives and parses the file containing the Instagram picture and creates the corresponding abstract action (Figure 6.7 step 2). The *Automaton Engine* maps the *open* Actions and translates the incoming "*ig*" file into a "*jpeg*" file (Figure 6.7 step 4). Finally, the *Composer* translates the abstract action into a file-call (Figure 6.7 step 4), which triggers the operating system to display to the user a list of co-located mobile applications that are able to handle the incoming image file via the *CloudMediator*, as for instance Dropbox (Figure 6.7 step 5).

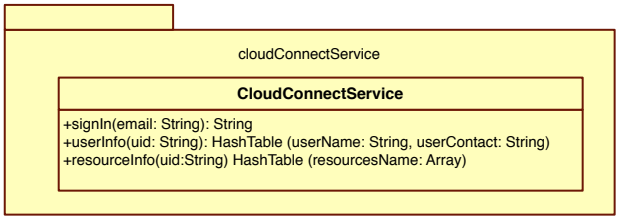


Figure 6.8: The Required Interface of the CloudConnect Application

Cloud CONNECTors

In order to experiment with App2Service CONNECTion, we consider the use of various Cloud storage services from a mobile platform. Indeed, with the growing usage of mobile applications, many companies provide Cloud services to help users access their content and synchronize data across their different devices, or interact and share content with other mobile users. The multitude of similar Cloud services, and the heterogeneity of their interfaces make it difficult for mobile applications to leverage these services, either directly or through the mobile applications of the Cloud services vendors. This issue is even worse when two interacting users intend to use different Cloud storage services.

We designed a demonstrator for the iOS platform that shows how a Cloud-enabled application can seamlessly interact with different Cloud services through mobile App2Service CONNECTors. The mobile application referred as *CloudConnect* requires a unique proprietary interface to interact with Cloud storage services. As illustrated in Figure 6.9, this interface defines the following actions:

- The *Sign-in Action* enables the application to login and authenticate the user
  - Name : signIn
  - Inputs: email
  - Output: uid (login ok), null (login fail)
- The *Get-user-info Action* retrieves user information that is registered in the corresponding cloud
  - Name: userInfo
  - Inputs: uid
  - Output: username, userContact
- The *Get-user-resources Action* lists all user resources in the related cloud
  - Name: resourceInfo
  - Inputs: uid
  - Output: Array of resourceName

We then designed four instances of the cloud CONNECTor to interconnect the application with major Cloud storage services, namely: Dropbox, Microsofts Skydrive, GoogleDrive and Flicker. All these CONNECTors *Register* to handle the *CloudConnect* interface and are able to map each *CloudConnect* action to its corresponding HTTP RESTful action of the CONNECTed Cloud services (Figure 6.9 shows the interfaces for each cloud service).

Each CONNECTor requires as input a mediator automaton to enable mapping between the required *CloudConnect* actions and the corresponding cloud service actions. The storyboard in Figure 6.11 shows the step-by-step interaction between the *CloudConnect* application and the Dropbox CONNECTor, and highlights how the three aforementioned Cloud storage actions are performed through the mobile CONNECTor. For instance, as the user pushes the authentication button, the application triggers a *signIn* action which displays a list of all co-located CONNECTors that were registered for the *CloudConnect* interface (Figure 6.11 step 1). The Dropbox CONNECTor is then selected by the user, and further receives the incoming action, which is parsed (by the MiAC Parser) and mapped (by the Automaton Engine) into a

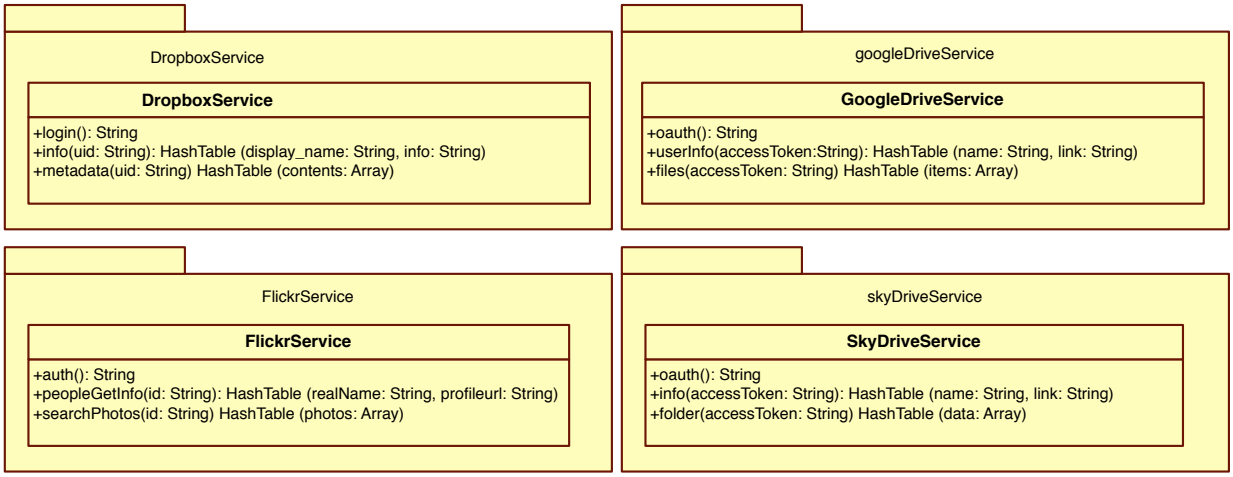


Figure 6.9: The Provided Interface of the Cloud Service

Dropbox action using the provided mapping (Figure 6.11 step 2). The CONNECTor authenticates users using OAuth[2] process. Then, if the user authorizes the caller application to access his account, the output of the action is filled and mapped back as a *CloudConnect* action before answering the caller (Figure 6.11 step 3). The *CloudConnect* application decodes the action and checks if the *signIn* action succeeded or not (Figure 6.11 step 4).

The demonstrator shows the deployment of more than one mobile CONNECTor on iOS, which is one of the most restrictive mobile platforms. We demonstrated how the in-house application was able to discover and interact with multiple legacy cloud services through the deployed CONNECTors.

From the development standpoint, when the CONNECTor skeleton is implemented, we substantially reduced the time of implementing all CONNECTor instances. As illustrated in Figure 6.10, the current CONNECTor generation achieved about three-quarters of the CONNECTor, which is already a satisfactory result. Still, we are now working on the integration with the Synthesis Enabler so as to generate the CONNECTor skeleton using the Synthesis Enabler.

6.6 Conclusion

Smartphones are now widely available and have taken over major markets over feature phones. Now, mobile users routinely install new applications, and use them not only to play but also to access online services or interact with others. As mobile applications rely more and more on online services or co-located applications for specific content or features, users will face the same heterogeneity challenges as on desktop and Web environments. In this chapter, we have presented the additional specific challenges that modern mobile platforms impose on collaborative applications that stem from: (i) the mobile applications marketplace model, which severely limits the dynamic delivery of applications, (ii) the highly fluctuating performance and reliability of mobile networks connectivity, and (ii) the runtime limitations imposed to mobile applications such as sandboxing.

While these constraints prevent the easy deployment of CONNECT Enablers on the mobile, or the usefulness of Enablers deployed in the networking environment, it remains possible to deploy CONNECTors on mobile devices to address the heterogeneity of the networked services available to mobile applications. Indeed, we have shown that mobile inter-application communication abstractions (such as Intents and Uri-schemes) can be used as an alternative means of communication for CONNECT, when deploying CONNECTors on mobile platforms. We have revisited the CONNECTor architecture with the necessary components to achieve interoperability with elements such as Registration, Local parsers and Local composers. Finally, we demonstrated the solution by developing a CONNECTor between the Instagram Legacy Application and four Cloud Storage services. As future work, we aim at further increasing the automation

[2]The OAuth 2.0 Authorization Framework: http://tools.ietf.org/html/rfc6749

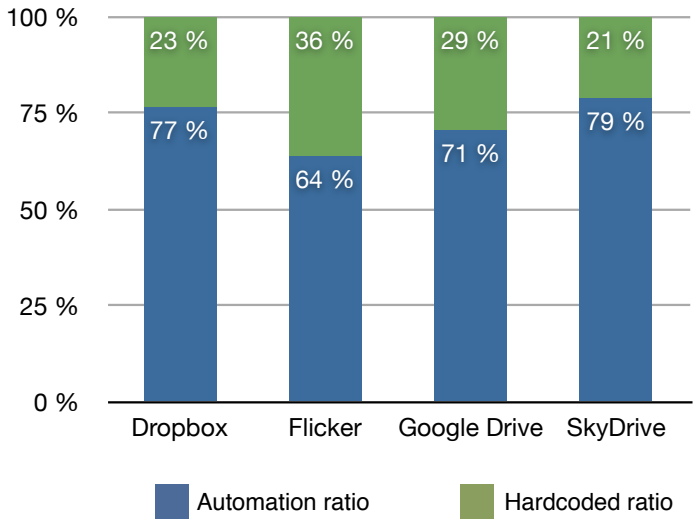


Figure 6.10: Automation Ratio for Generating the Cloud CONNECTors Instances

ratio. This can be achieved by integration with the Synthesis Enabler to automatically produce the mediator automata. The generation of local and network, parsers and composers can also be improved by using a model-driven generation approach, either integrating with Starlink or the FCCL framework (see Deliverable D3.4 for the latter).

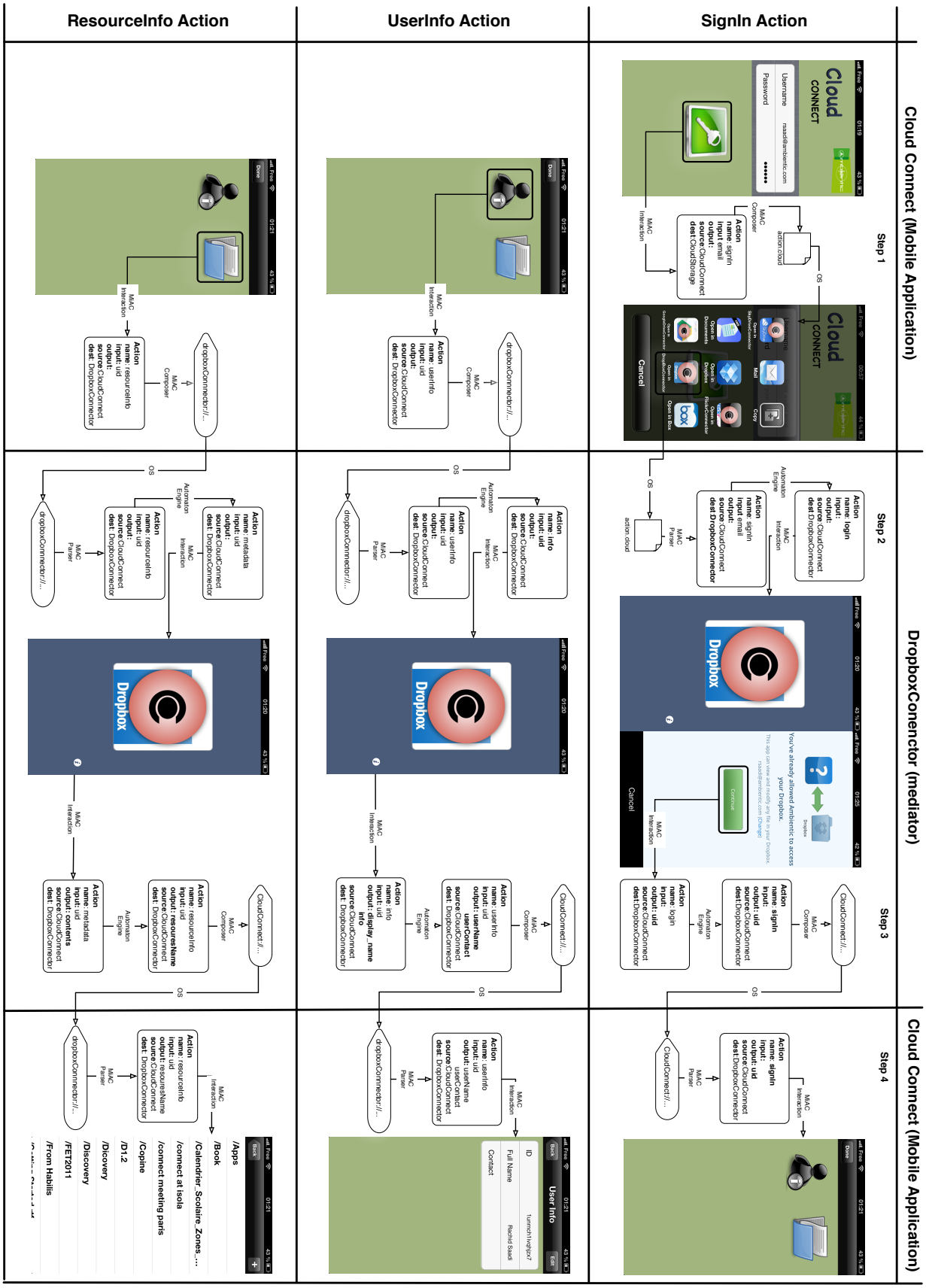


Figure 6.11: Cloud CONNECTion Storyboard

7 Evaluation

7.1 Introduction

In this section we perform the assessment of the CONNECT architecture for the two assessment criteria originally proposed in Deliverable D6.3 [28]:

- Modelling and reasoning about peer system functionality (see Section 7.2).
- Connecting eternal systems (see Section 7.3).

Specifically in Section 7.3, we perform evaluation of the CONNECT architecture in terms of it achieving the required functionalities in a number of case studies. These case studies exhibit heterogeneity in a number of dimensions that pose challenges to achieving interoperability. We show that the CONNECT architecture can successfully produce the correct CONNECTor for each case and deploy this dynamically. We also show that the performance of the produced CONNECTor is acceptable to the networked systems employed in the cases. Based upon these experiences, we also evaluate the CONNECT architecture against the state of the art in distributed systems solutions for achieving interoperability, and show that CONNECT far exceeds the abilities of other platforms.

Originally in Section 5.2.6 of Deliverable D6.3 [28]: *Connecting eternal systems* three objectives were listed:

- Objective 1: To utilise the CONNECT architecture to automatically generate and deploy concrete CONNECTors that successfully ensure correct interoperation between two matched networked systems.
- Objective 2: To correctly adapt a deployed CONNECTor when the non-functional requirements are validated or the initial CONNECTor is incorrect with respect to the functional requirements.
- Objective 3: For the technologies and solutions created within the CONNECT to have a broader impact in the field of distributed systems; and for the wider use of CONNECT software to resolve interoperability problems.

However, as described in the previous chapters the role of adaptation in the CONNECT architecture remains conceptual, and without practical implementation the required evaluation of Objective 2 could not be effectively carried out. Hence, the following subsections concentrate on Objective 1 and Objective 3 respectively.

7.2 Modelling and reasoning about peer system functionalities

7.2.1 Objective 1: To accurately model the interfaces and functional behaviour provided and required by a given networked system.

To face the interoperability challenge arising in increasingly complex distributed systems, it is important to have available networked system specifications that allow us to reason, automatically, about the functional and behaviour semantics of networked systems. To that end, we take inspiration from semantic Web services as they advocate semantically-annotated models for networked systems. We significantly add to the related state of the art by in particular addressing: (i) behavioural specification for networked system that builds on eLTS for automated reasoning and enactment of behavioural matching, (ii) specification of protocols at both application and middleware layers to allow the synthesis of mediators that solves behavioural mismatches at both layers .

Concerning the latter point, we defined an ontology for middleware and described the process of using it to generate mediators that solve interoperability from middleware to application. The translation of concrete interface description, including middleware primitives, into middleware-agnostic actions can be performed automatically so as to allow us to reason about interoperability at an appropriate level of

abstraction. The concretisation is however more complex. Using Starlink the binding from middleware-agnostic operations to network-level messages is specified using an appropriate DSL. We are also investigating a compositional technique to the generation of cross-layer parsers and composer, which gave us promising results (see Deliverable 3.4). This work is being further investigated by Inria within the CHOReOS project [20].

However, while the description of networked systems using ontology-based annotations and behavioural specification has been acknowledged as necessary for automating interoperability assurance, it is often the case that networked systems exhibit only their syntactic interface. To overcome the lake of rich networked system specifications, we used advanced learning techniques to infer the affordance and the behaviour of NSs automatically. The assessment of behavioural learning is reported in Deliverable 4.4 [16] while that of affordance learning was in Deliverable 1.3 [3].

7.2.2 Objective 2: To match two networked systems correctly.

Having shown that automatic inference of affordance on the basis of interface descriptions is indeed feasible, we must also show that the affordances learnt result in the expected benefit to discovery. The purpose of introducing affordances is to filter the number of service pairs for behavioural matching with a relatively efficient semantic check, and hence to reduce the overall time taken to conduct matchmaking when services are discovered.

After performing training offline, we integrated the trained classifier into the Discovery Enabler, which is responsible for matching pairs of networked systems. The Discovery Enabler invokes the classifier when it discovers a networked system that does not have an affordance. We then measured the time taken by the Discovery Enabler to perform matchmaking with and without the classifier.

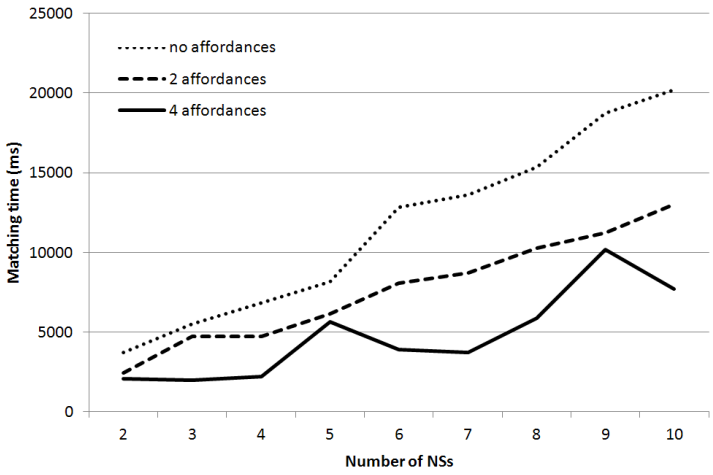


Figure 7.1: Performance of matching with 0, 2, and 4 affordances.

Figure 7.1 shows the time taken to perform matchmaking after the sequential discovery of the given number of networked systems (up to 10). The results are averaged over ten runs. The line with the steepest average gradient shows the time taken when no affordances are used, and so no categorisation takes place. Matchmaking in this case involves performing behavioural matching for every possible pair, i.e. n^2 checks for n NSs. The other lines show the time taken when the services are automatically categorised into two and four affordances respectively. Having just two affordances reduces the number of behavioural checks to $\frac{n^2}{2}$ and adds n^2 semantic checks. In the results, we find two affordances gives a 32% reduction in the matching time, and four affordances gives a further 37% reduction.

When two or three systems have been discovered, in the case with four affordances, we do not yet expect any matches. In fact the results show an almost constant time, around 2 seconds, for matching when no matches are found. This delay represents the overhead inherent in our prototype implementation of discovery resulting from parsing WSDL and BPEL and other steps internal to discovery.

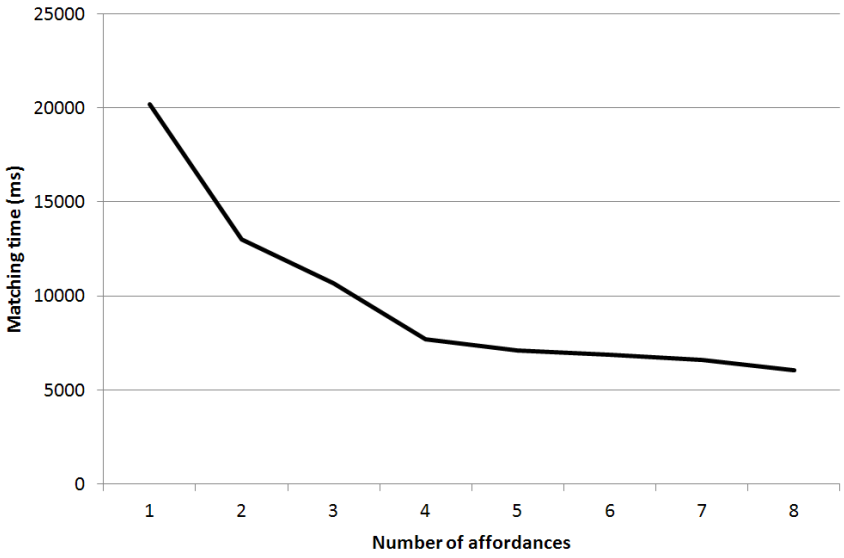


Figure 7.2: Performance of matching after discovering 10 networked systems.

Figure 7.2 shows the reduction in matching time as the number of affordances increases towards the number of systems. It can be observed that the worst case time involves one affordance or none, and the best case involves as many affordances as there are networked systems (no semantic matches will be found and so no behavioural checks will be required). This suggests that the domain ontology (taxonomy) in which the affordances are defined should be as detailed as possible. Note however that increasing the number of affordances can decrease the accuracy of categorisation as features (tokens in interface descriptions) become increasingly ambiguous. This effect can be seen to an extent in the second categorisation experiment with 10 categories compared to 2 categories in the first experiment.

Through the application of support vector machines for text categorisation, we have shown that the burden of categorising systems, that is, determining their high-level functional semantics, can be lifted from the engineer and performed automatically with reasonable accuracy. The cases of inaccuracy can be divided into false positives, where two NSs have been assigned the same affordance when in fact they do not match, and false negatives, where two matching NSs are assigned different affordances and hence no attempt to connect them will be made. Minimising the number of false negatives (i.e. maximising recall) is hence critical for CONNECT. Greater accuracy may be achieved by finding more nuanced features, such as the structure of the document or token proximity, on which to base the categorisation.

Given such categorisation, affordance matching allows us to reduce the number of behavioural checks performed, and thus increase the performance of the matchmaking process as a whole. Our results show that the gain is relative to the number of affordances, with just two affordances providing a 32% performance increase. This performance increase benefits our overall aim in the CONNECT project, which is to provide solutions for interoperability at runtime, thus requiring efficient runtime mechanisms to identify compatibility and find solutions for overcoming incompatibilities.

Automatic categorisation has been studied previously in work such as [24, 30]. The latter assigns semantic concepts to web services by considering their WSDL descriptions but without taking into account the unstructured data potentially available within the documentation tag that can give more information about the category the web service belongs to. Instead of attaching a category concept to a web service, Klusch *et al.* [29] propose to evaluate the similarity between a pair of web services based on both structured and unstructured information included in their interfaces using support vector machines. This approach is the closest to ours but is clearly not scalable especially when considering environments where services may continuously be discovered.

Concerning the accuracy of behavioural matching, since only the mapping-based synthesis was retained (see Deliverable 3.4 [18]), there are no false positive behavioural matches as long as the behavioural models of the networked systems are correct, i.e., they represent the actual behaviour of each NS.

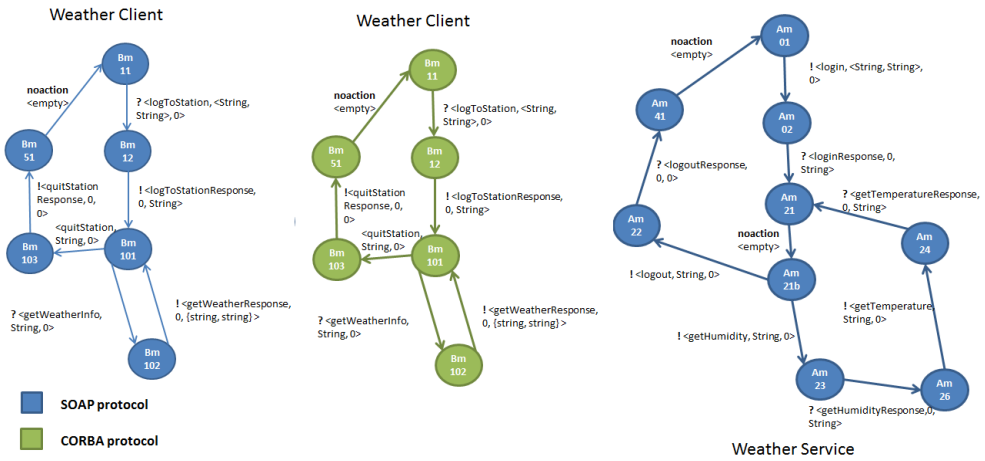


Figure 7.3: Generated concrete coloured automata for the weather client and service networked systems both implemented using SOAP

7.2.3 Objective 3: To perform CONNECT networked system modelling and matching in a performant manner.

The performance of the modelling relates to that of the automated learning the affordances and behaviours in order to complete the specification of NSs and are reported in Section 7.2.1. The performance of affordance matching are described in 7.2.2 while that of behavioural matching are reported in Deliverable 3.4 [18].

7.3 Connecting eternal systems

7.3.1 Objective 1: To utilise the CONNECT architecture to automatically generate and deploy CONNECTors

Case Study 1: Weather Services

In this case study, three networked systems exist: i) a weather service implemented using SOAP providing temperature and humidity information, ii) a client application for retrieving weather information implemented using SOAP, and iii) a client application for retrieving weather information implemented using CORBA. With these networked systems operating, the CONNECT architecture (specifically the CONNECTion phase) was deployed. This discovered, learned and generated the models of the networked systems behaviour and the concrete coloured automata of two required CONNECTors were synthesized and deployed in the network to ensure that the networked systems were able to fully interoperate as follows:

- The SOAP client whose coloured automata specification is illustrated in Figure 7.3 correctly interoperated with the SOAP weather service whose coloured automata is provided in Figure 7.3. The synthesized concrete mediator that correctly performs interoperation is shown in Figure 7.4.
- The CORBA client whose coloured automata specification is illustrated in Figure 7.3 correctly interoperated with the SOAP weather service whose coloured automata is provided in Figure 7.3. The synthesized concrete mediator that correctly performs interoperation is shown in Figure 7.5.

This case study highlights how the correct CONNECTor is produced in the face of different heterogeneity dimensions:

1. Behavioural mismatches. The SOAP client and the SOAP server differ in terms of the action sequences. The client performs a single operation `getWeather` that returns a data record with two

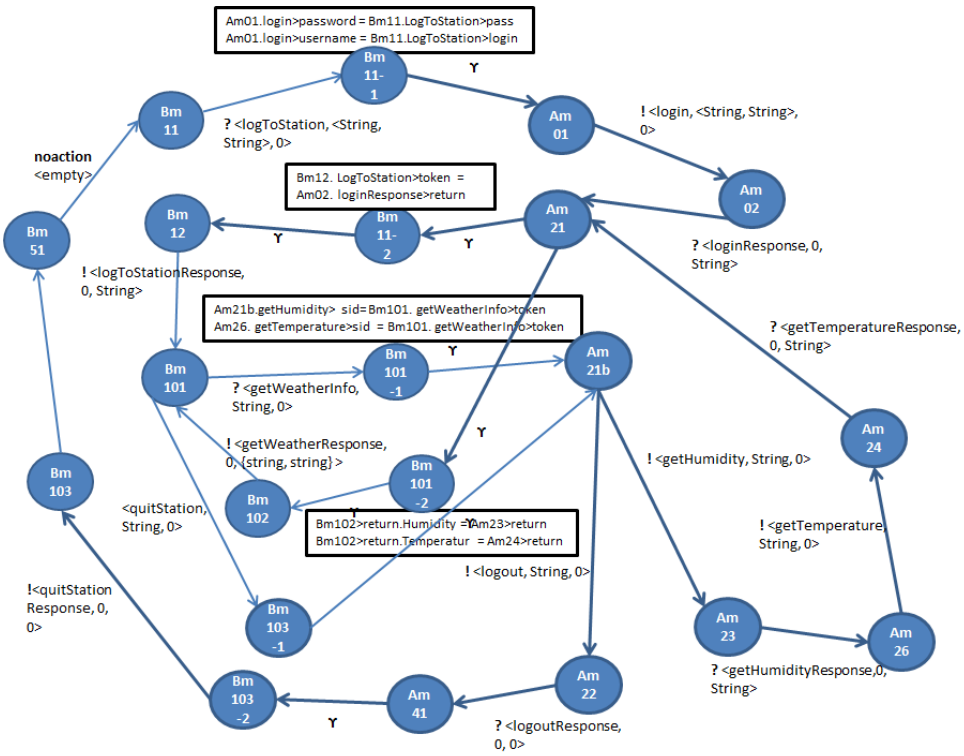


Figure 7.4: Generated coloured automata mediator for the weather case study (SOAP to SOAP)

string: `temperature` and `humidity`. Whereas, the server provides two separate operations: `getTemperature` and `getHumidity` that both return a single string value. The synthesized CONNECTor correctly calls both operations to inform the return value of the client action.

2. Data mismatches. In the prior example, the results are returned by the server in separate parameters, whereas the required data format for the result is a structured data element of two fields. The CONNECT architecture is able to synthesize the correct translation logic in the concrete CONNECTor in order to resolve the data mismatch.

3. Protocol mismatches. The two clients are implemented with distinct RPC protocols (CORBA and SOAP) that differ in network semantics, data types, packet format, and message fields. Importantly, one is binary and the other is XML-based. The CONNECT architecture is able to generate the correct CONNECTor in each case and bind this correctly to the required middleware protocols such that they can execute correctly.

Case Study 2: Positioning Services

In this case study, two networked systems exist: i) a positioning service implemented using AMQP that advertises the location of systems within a geographic area , ii) a client application for retrieving positioning information requested systems; this was implemented using the SOAP protocol. With these networked systems operating, the CONNECT architecture (specifically the CONNECTion phase) was deployed. This discovered, learned and generated the models of the networked systems behaviour and the concrete coloured automata of two required CONNECTors were synthesized and deployed in the network to ensure that the networked systems were able to fully interoperate as follows

- The SOAP client whose coloured automata specification is illustrated in Figure 7.6 correctly interoperated with the AMQP position service whose coloured automata is provided in Figure 7.6. The synthesized concrete mediator that correctly performs interoperation is shown in Figure 7.6.

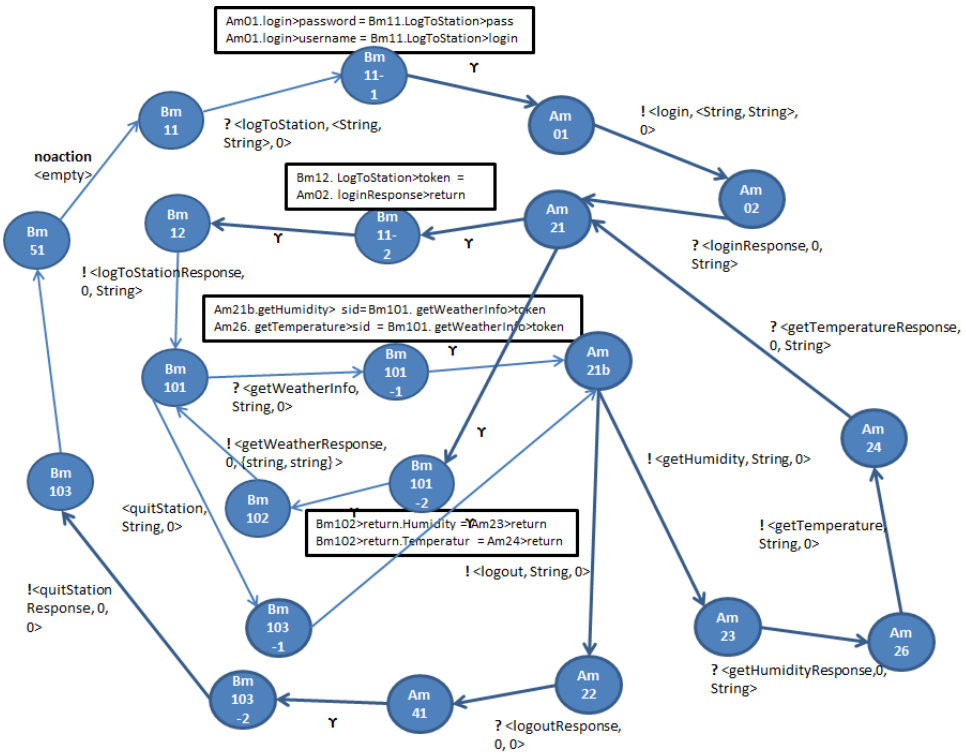


Figure 7.5: Generated coloured automata mediator for the weather case study (CORBA to SOAP)

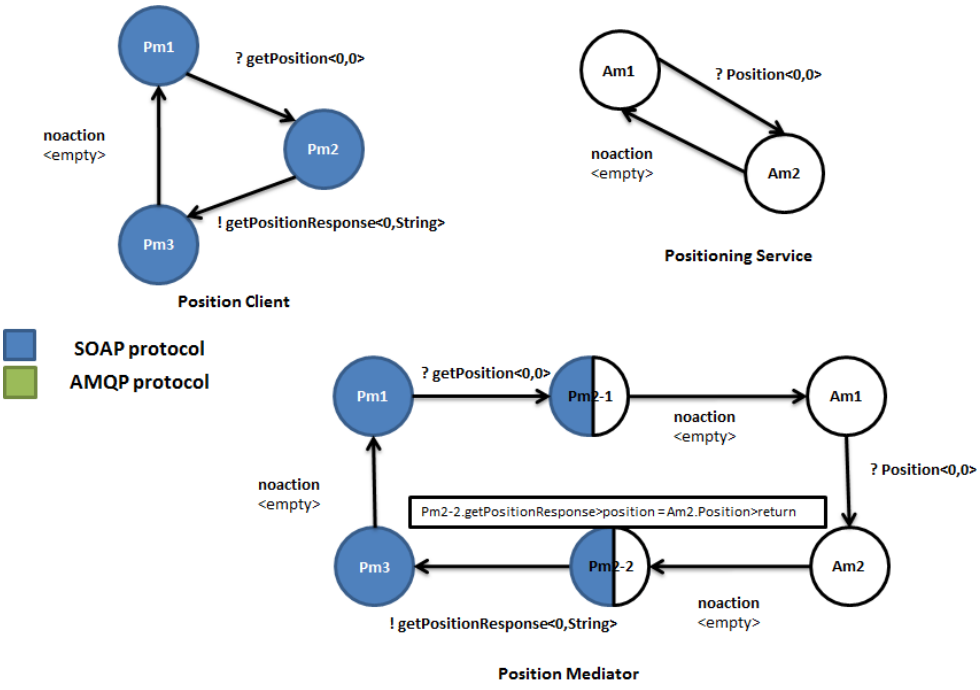


Figure 7.6: Generated coloured automata mediator for the position case study (SOAP to AMQP)

The position case study highlights how the correct CONNECTor is produced in the face of different heterogeneity dimensions:

1. Behavioural mismatches. The two networked systems employ very different behaviour styles. One

is a request response protocol to request positions, whereas the position service is implemented using a publish-subscribe abstraction to periodically and asynchronously publish positions. The use of coloured action automata is able to abstract over these differences to generate a functioning mediator that can then be bound to the middleware protocols to resolve their differences as described next.

2. Protocol mismatches. The two clients are implemented with distinct middleware protocols (AMQP and SOAP) that differ in network semantics, data types, packet format, and message fields. Importantly again, one is binary and the other is XML-based. However, there is a large difference in the protocol behaviour-SOAP is synchronous request response, whereas AMQP is asynchronous message publication to a single channel. However, in spite of the differences the CONNECT architecture is able to generate the correct CONNECTor in each case and bind this correctly to the required middleware protocols such that they can execute correctly.

Analysis of Case Studies

The original aim was to evaluate whether the CONNECT architecture achieved the objective to automatically generate and deploy the correct CONNECTors so that the networked systems interoperate effectively. We previously introduced heterogeneity dimensions that hinder interoperability. These case studies illustrate that CONNECT can address these dimensions: *application data* i.e., in terms of the different types of data structure and the ability to create translation logic to map between; *application behaviour* i.e., synthesis can produce CONNECTors that handle sequence mismatches and other behavioural differences; and *middleware heterogeneity*, i.e., Starlink is able to bind concrete automata to heterogeneous middleware protocols (RPC and publish-subscribe) and ensure the mapping between different middleware protocols is executed correctly.

Overall, the results from the case studies show that the architecture was able to correctly perform in order to generate a CONNECTor that once deployed transparently ensures that the networked systems interoperate correctly. Hence, a key objective of CONNECT has been met, i.e., that in the face of extreme heterogeneity, interoperability solutions can be created on-the-fly. These case studies and results are explored further as part of the demonstrator and results provided in deliverable D6.4 [14].

Performance evaluation

Using the previous case-studies we measured the throughput performance of the generated CONNECTors and then compared them to the benchmark performance of a client communicating with the service through the required middleware platform. For example, where a corba client communicates with a SOAP service, the benchmark is a corba client communicating with a service using CORBA (as this the expected throughput of the client).

To perform the experiment, all networked systems were deployed on the same host as the CONNECTor. The throughput (operations/sec) was measured on the client by performing 1000 operations after an initial warm up of 50 operation calls. Each test case was performed 10 times and the median value is reported here. A desktop PC with an Intel Core Duo CPU @ 3Ghz, 4 Gb RAM, and running Windows 7, with Java version 1.7. The results of the tests are illustrated in Table 7.1.

Table 7.1: Implementation and integration progress of CONNECT enablers for Connection

Networked System	with CONNECTor throughput (ops/sec)	without CONNECTor benchmark throughput (ops/sec)
Weather Service SOAP	114	236
Weather Service SOAP with monitoing	103	NA
Weather Service CORBA	350	1275
Positioning Service AMQP	320	1657

The results in the table show that there is percentage decrease in throughput when the CONNECTor is deployed. This is to be expected as the introduction of a CONNECTor cannot compete with an optimized middleware platform. However, for the most case this remains acceptable - none of the running systems

timed out during an operation call. Furthermore, for the case of monitoring being introduced into the CONNECTor it can be seen that there is only a small 5 percent decrease in throughput. The original objective was to assess if generated CONNECTors perform in a performant manner, and while there is an expected overhead this does not hinder the operation of the networked systems.

7.3.2 Objective 3: To have impact in the field of Distributed systems and to resolve interoperability problems beyond SOTA

In Deliverable D1.1 [5] we provided a comprehensive survey of the state of the art. The results of this state of the art are provided in table 7.7. This survey highlighted two important concerns; first, there is a clear disconnect between the main stream middleware work and the work on application, data, and semantic interoperability; second, none of the current solutions addresses all of the requirements of dynamic pervasive systems. Here we leverage the results from CONNECT to show that these concerns have been resolved by the project, and our work significantly goes beyond the state of the art.

Figure 7.7 shows that no solution attempts to resolve all five dimensions of interoperability. However, it is important to identify that CONNECT does resolve all five dimensions of interoperability:

- *Discovery*. The discovery enabler addresses the problem of networked systems using heterogeneous discovery methods and protocols. The derivation of common models of networked system behaviour abstracts this heterogeneity and allows systems to matched such that they can interoperate in the future.
- *Interaction*. The synthesis enabler can generate application action coloured automata that abstract from the middleware interaction type (e.g. RPC, pub-sub, etc.). The CONNECTor architecture (and the Starlink framework) can then bind these to the heterogeneous middleware protocols. Indeed we showed in the prior case studies how different middleware protocols of the same style (e.g. RPC protocols: SOAP and IIOP) interoperate; but more importantly, how middleware protocols of different styles, e.g. the AMQP publish-subscribe protocol and the SOAP RPC protocol, interoperate successfully.
- *Data*. Bridging transitions in Coloured automaton support the translation and functional transformation of data between actions in the networked systems. For example, in the first case study the angle parameter can be changed from degrees to radians. Hence, the underlying Starlink tool supports the necessary mechanisms to support full data interoperability.
- *Application*. The synthesis method is able to match and map application operations based upon semantic equivalences between two networked systems. Hence, where there are behavioural mismatches, synthesis can generate a concrete CONNECTor that ensures interoperability.
- *Non-functional*. We have demonstrated that CONNECTors can be evaluated against non-functional properties, monitored at runtime, and also instrumented to maintain non-functional properties, e.g., in the case of security-based policy enforcement.

Further, the transparency column shows that only the transparent interoperability solutions achieve interoperability transparency between all parties (however only for a subset of the dimensions). The other entries show the extent to which the application endpoint (client, server, peer, etc.) sees the interoperability solution. ReMMoC, UIC and WSIF rely on clients building the applications on top of the interoperability middleware; the remainder rely on all parties in the distributed system committing to a particular middleware or approach. Further, the abstraction column demonstrates that (at some level) conformance to a particular abstraction is required to achieve interoperability. That is, IDL and WSDL are the abstractions that applications are developed with; or different middleware are mapped to a common abstraction independent of the application e.g. the service discovery models and descriptions for the transparent interoperability solutions. These themselves are typically focused to a particular abstraction type e.g. service-orientation; hence no solution covers the full diversity of communication abstractions e.g. making tuple spaces, message-based, RPC and publish-subscribe systems interoperate in a spontaneous transparent fashion. CONNECT on the other hand is a transparent solution with no conformance to a particular abstraction (as we have previously argued). However, rather than a conformance to a particular

	SD = Discovery I = Interaction D= Data A = Application N=Non-functional						
	SD	**I**	**D**	**A**	**N**	**Abstraction**	**Transparency**
CORBA		X				IDL Interfaces	CORBA for all
Web Services		X				WSDL Services	WSDL & SOAP for all
ReMMoC	X	X				WSDL Services	Client-side middleware
UIC		X				RPC	Client-side middleware
WSIF		X				WSDL Services	Client-side middleware
MDA		X				Soft. Arch.	Platform Independent models
UniFrame		X				Soft. Arch.	Platform Specific models
ESB		X				Message bus	Bridge connector
MUSDAC	X					Service desc.	Connection to middleware
INDISS/ Nemesys	X					Service desc.	Yes
uMiddle	X	X				Service desc.	Yes
OSDA	X					Service desc	Yes
SeDiM	X				X	Service desc	Yes
SATIN	X	X				Mobile components	Choice of SATIN for all
Jini		X				Mobile proxies	Choice of Jini for all
Semantic Middleware			X	X		Service desc. With semantic information	Choice of same semantic middleware for all
Semantic Web Services		X	X	X		WSDL Services	WSDL for all plus commitment on a semantic framework and ontologies
CONNECT	**x**	**x**	**x**	**x**	**x**	**No fixed abstraction**	**Yes**

Figure 7.7: Evaluation summary of effectiveness of the state of the art against each interoperability dimension

technology CONNECT is tied to higher reasoning concepts, e.g. the availability of semantic information about the systems that is then returned via discovery. However, we believe such a coupling is much more effective than with standards and technologies, in so much that there is the potential to learn and build ontological information about an unknown system, and there is a growing and readily available base of global ontologies to support this process.

Given the ability of CONNECT to address all of these dimensions it is clear that CONNECT has the potential to have a significant impact on the state of the art in both middleware and distributed systems. Importantly, with respect to the problem of the divergent middleware and semantic web communities, who evolved independently - these results indicate a pathway for the technologies of both camps to be merged together. Indeed, the publication of a ”Big ideas” paper at Middleware 2011 [7] by the CONNECT project is clear indication of the willingness to merge the richness of the work performed on both sides.

The initial impact of the CONNECT project is already growing. In addition to the Big Ideas paper, the CONNECT architecture work has also been published at leading distributed systems venues: a paper de-

scribing Starlink [10] was published at ICDCS 2011 and a paper about concrete application automata [9] was published at Middleware 2011. Furthermore, numerous keynotes presenting the CONNECT architecture and technologies have been presented at distributed systems conferences and workshops. Based upon this early evidence, there is increasing awareness of the work of CONNECT and its potential to influence the field of interoperability.

8 Conclusions

8.1 Concluding Remarks

The overall aim of the CONNECT project was to address the interoperability challenges that result from the use of different data and protocols by the different entities involved in the software stack such as applications, middleware, platforms, etc. This is to be performed on-the-fly in order to succeed in highly heterogeneous, dynamic environments where systems must interact spontaneously i.e. they only discover each other at runtime.

This deliverable has presented the final version of the CONNECT architecture, and in particular reported on both the finalised architecture specifications and implementations. Importantly this highlights the conclusion of the integration work carried out during the the fourth year of the project. On the whole this has been a successful venture; a set of diverse technologies from the individual work packages have been brought together by the common architectural framework to realise on-demand, and on-the-fly interoperability. Here it can be seen that the connection-time process from discovery through to deployment is well integrated and evaluated, while the integration of CONNECTability behaviour has progressed significantly. This is highlighted by the use of the CONNECT architecture to realise the demonstrator scenario presented in Deliverable D6.4 [14]. Further, a number of prototype software solutions have been created to complement the realisation of the CONNECT architecture. Those highlighted in this deliverable and described in the accompanying appendix (Deliverable D1.4 Appendix - Prototype).

In the fourth year, the key achievements include:

- The enhancement of the concrete k-coloured automata CONNECTors to integrate with the behaviour of the enablers providing non-functional properties. Importantly, policy-based security was successfully integrated into the concrete CONNECTors.
- A systematic evaluation of the automatically generated CONNECTors to show that they correctly interoperate with concrete networked systems, and do so in a performant manner.
- Identification of the important contributions CONNECT has made to the state of the art in middleware and distributed systems.
- An exploration of the long-lived management of the enabler architecture, and the dynamic adaptation of CONNECTors using model-based reflective solutions.

8.2 Future Directions

Given the success of CONNECT in identifying new solutions to the interoperability problem, potential future directions for the work must concentrate on the impact that these solutions can have on the state of the art.

One of the key contributions of the CONNECT architecture has been the use of software models to create concrete CONNECTor solutions. However, this has so far focused on simple two party communication patterns, and within simple application domains. We believe that there is huge potential to apply the philosophy of k-coloured automata and the associated Starlink tool to a much broader range of distributed systems problems. For example, work is already ongoing in a number dimensions regarding external uptake of the tools:

- *Mobile and Cloud Computing*. This deliverable has discussed how Ambientic has leveraged the CONNECT architecture to develop interoperable media streaming solutions for the mobile application domain. Further, the ideas of the CONNECT architecture are also being applied to the domain of mobile interaction with Cloud services. Ambientic will continue in this direction with regards to their iBICOOP Middleware solutions.
- *Environmental Observatory*. A environmental science project funded by the NERC UK government agency is investigating the use of cloud computing to underpin the often complex, and heterogeneous sytems that are used to model and predict environmental phenomenon. Within this project,

Lancaster University is investigating how k-coloured automata can support a cloud broker that handles the migration of data and functionality between heterogeneous cloud providers.

- *OverStar*. Multi-party distributed systems, e.g. P2P and overlay network based systems are highly heterogeneous and themselves face complex interoperability problems. The University of Bordeaux and Lancaster University are building model-based overlay network software to address these challenges, using Starlink as the starting point. This work has already resulted in a publication at Middleware 2012 [21].
- *Semantic Interoperability*. One of the key results of the CONNECT architecture is the use of semantic technologies to underpin interoperability and middleware solutions. Partners across the project continue to leverage these results in new work.
- *Models@runtime*. CONNECT is heavily based around software models that are available at runtime. PKU is working on a number of software engineering projects focused on models@runtime using the CONNECT project as an application area. With this they continue to improve their SM@RT tool.

9 D1.4 Appendix: Published Papers

Adaptation remains an open concern within the CONNECT architecture. Conceptual mechanisms have been put forward both in this Deliverable, and in Deliverable D5.4; however, this have not been fully implemented. Hence, we present additional collaborative work from the project partners that has focused on adaptation specifically related to the dynamic adaptation of CONNECTors. This takes the form of two papers which provide insight into future directions for runtime management of CONNECTors. A third paper also explores the work on advanced learning within the project:

- Paper 1: N. Bencomo (INRIA), A. Bennaceur (INRIA), P. Grace (LANCS), G. Blair (LANCS), V. Issarny (INRIA), The Role of Models@run.time in Supporting On-the-fly Interoperability. Springer Journal on Computing (to be published 2013).
- Paper 2: Y. Ma (PKU), X. Liu (PKU), G. Huang (PKU), P. Grace (LANCS), Model-based Management of Service Composition. To appear in SOSE 2013: 7th International Symposium on Service Oriented System Engineering, March 25 - March 28, 2013, San Francisco, USA.
- Paper 3: A. Bennaceur(INRIA), V. Issarny (INRIA), D. Sykes (INRIA), F. Howar (TUD), M. Issberner (TUD), B. Steffen (TUD), R. Johannson (Trento), A. Moschitti (Trento), Machine Learning for Emergent Middleware. To appear in Springer Communications in Computer and Information Science (CCIS)

Noname manuscript No.
(will be inserted by the editor)

The Role of Models@run.time in Supporting On-the-fly Interoperability

N. Bencomo · A. Bennaceur · P. Grace · G. Blair · V. Issarny

Received: date / Accepted: date

Abstract Models at runtime can be defined as abstract representations of a system, including its structure and behaviour, which exist in tandem with the given system during the actual execution time of that system. Furthermore, these models should be causally connected to the system being modelled, offering a reflective capability. Significant advances have been made in recent years in applying this concept, most notably in adaptive systems. Our hypothesis is that a similar approach can also be used to support the dynamic generation of software artefacts at execution time. An important area where this is relevant is the generation of software connectors to tackle the crucial problem of interoperability in distributed systems. We refer to this approach as emergent middleware, representing a fundamentally new approach to resolving interoperability problems in the complex distributed systems of today. In this context, the runtime model is used to capture meta-information about the underlying networked systems which need to interoperate including their interfaces and additional knowledge about their associated behaviour. This is supplemented by ontological information to enable semantic interoperability in given application domains. This paper focuses on this novel use of models at runtime, examining in detail the nature of such runtime models coupled with consideration of the supportive algorithms that extract this knowledge and use it to synthesise the appropriate emergent middleware.

Keywords runtime interoperability, mediators, ontology, runtime models

N. Becomo, A. Bennaceur and V. Issarny
Inria, Paris-Rocquencourt, France
E-mail: firstname.lastname@inria.fr

G. Blair and P. Grace
School of Computing and Communications, Lancaster University, UK
E-mail: {gracep,gordon}@comp.lancs.ac.uk

1 Introduction

A *model@run.time* or runtime model can be defined as an abstract representation of a system, including its structure, behaviour and goals, which exists in tandem with a given system during the actual execution time of that system. Furthermore, this model should be causally connected to the system being modelled, hence offering a reflective capability. Causal connectivity allows the runtime model to provide up-to-date information about the system in order to support analysis of the system before committing to changes, and therefore avoiding potential inconsistencies in the runtime system. As the system evolves, the runtime model should also evolve to embody the new state.

Runtime models differ from development-time models in several ways. In contrast to development-time models that usually convey software design meaning, runtime models represent abstractions of runtime phenomena from a problem space perspective [20]. Runtime phenomena are affected by environmental conditions and contexts found during execution. Crucially, it is difficult to assess accurately the impact of the changes in the environment and context before deployment and runtime due to incomplete information. Runtime models support the handing of these dynamic and to some extent unforeseen changes [46]. Importantly, runtime models can support decision making and reasoning based on knowledge unforeseen prior to the time of execution, but which emerges during execution.

Significant advances have been made in the use of runtime models [6, 4]. Architectural-based runtime models is a research topic that has generated most interest [37, 36, 16, 48]; and mainly in the broader area of self-adaptation [36, 32, 22, 33].

Currently, there is pressure to move some activities from design-time to deployment and runtime [3]. One of the goals is to be able to insert at runtime new behaviour that was not necessarily foreseen during design-time. One way to do this is to be able to synthesise the software associated with the new behaviour at runtime. As self-representations of the systems (as with traditional MDE [40]), runtime models can be used as the basis for software synthesis. However, little attention has been directed to techniques for synthesis or generation of software using runtime models during execution. This is precisely the topic we aim to address in this paper. We argue that runtime models can support the runtime synthesis of software that will be part of the executing system and which was not necessarily conceived during design time.

In modern highly dynamic environments, devices appear and disappear along with the services they offer. Where they meet spontaneously, interoperability is a fundamental requirement. These services may not know each other, but they may still try to interact in order to meet certain goals [3]. Therefore, it may be the case that for some aspects of the system, a software model needs to be conceived during runtime as it would be impossible to design it in advance. Inferring information to create runtime models [41] during execution using for example learning techniques [5, 43] offers an interesting approach to take.

In this paper, we focus on the novel use of runtime models to support the synthesis of emergent middleware, i.e., the synthesis of mediators that translates actions of one system to the actions of another system developed with no prior knowledge of the former in order to achieve interoperability. Using rich discovery and learning methods we are able to capture and refine the required knowledge of the context and environment. By means of reification, the knowledge is explicitly formulated and made available to computational manipulation in the form of a runtime model. This runtime model is based on labelled transition systems (LTSs) [28] which offer the behavioural semantics needed to model the interaction protocols. Ontologies complement the LTSs providing semantic reasoning about the mapping between protocols. From these runtime models, mediators are synthesised. Supported by new acquired knowledge, new versions of the runtime are obtained and therefore a new mediator can be synthesised accordingly.

In summary, the contribution of this paper is an approach to synthesise software, in the form of mediators, from runtime models. The core piece of this novel approach is the derivation of completely new runtime models during execution to solve the on-the-fly interoperability problem i.e. creating a mediator from scratch. Crucially, the runtime models capture not just structure and functionality but also behaviour which is refined using machine learning techniques. We also use ontological information to support conceptual reasoning based on models.

The paper is structured as follows. In section 2, we discuss in detail the interoperability problem in complex distributed systems. In section 3 we present how models at runtime are used to dynamically synthesise the emergent middleware that ensure interoperation between heterogeneous networked systems. In section 4 we discuss some related work. Finally, we draw conclusions in section 5.

2 Emergent Middleware to support On-the-fly Interoperability

2.1 The Need for On-the-fly Interoperability

Interoperability is defined as the capability of two or more networked systems to exchange and understand one another's data. Where systems are designed and developed with knowledge of one another, or where systems have been developed using a common standard then the interoperability problem is largely solved. Indeed, interoperability is a primary goal of standard-based middleware solutions (e.g. CORBA and Web Services middleware) have been successfully utilised to achieve interoperability. However, the increasing complexity of distributed systems introduces new problems, which existing middleware-based interoperability approaches are not suited to address.

In environments where heterogeneity and highly dynamic behaviour are typical, e.g. pervasive computing, mobile computing, and large scale systems of system, there are further challenges to achieving interoperability. Hetero-

geneity can be encountered in many forms. Middleware is often applied to address differences in terms of computational devices, communication networks, and operating systems, however, the design decisions taken for the development and deployment of each networked system are potential interoperability bottlenecks. Using a particular middleware type means that interoperability is not possible with networked systems employing other middleware solutions due to the differences of the communication protocols. Further, differences in the design of the application interface will hinder interoperability, hence even where a common middleware is chosen interoperability cannot be guaranteed. Differences in the syntax of interfaces, the types and data formats, the semantic meaning of data schemas, and the invocation sequence required for achieving application functionality are all potential problems. Dynamic behaviour is characterised by systems and services that come and go (often due to the increasing mobility of users); furthermore, the operating conditions in heterogeneous environments fluctuate, e.g., changing quality of service levels in mobile networks. Interactions are also spontaneous, i.e., systems wanting to interoperate, search at runtime for systems that match their requirements. Here there can be no agreement of a common solution or standard, and the differences in application behaviour and communication protocols can only be detected and resolved at runtime.

Therefore, it is difficult to design a solution that takes into account the many dimensions of heterogeneity, and this is further exacerbated by spontaneous interactions and so no prior decisions about interoperability solutions can be assumed. Instead, a fundamental rethink is required into how interoperability can be resolved at runtime without relying on common standards or design decisions. We argue that models@runtime have an important role to play in such solutions.

2.2 The GMES Case Study

To better illustrate the interoperability problem we highlight the challenges through the use of a case study in the area of the Global Monitoring for Environment and Security (GMES[1]). GMES is the European Programme for the establishment of a European capacity for Earth Observation. In particular, the emergency management thematic highlights the need to support emergency situations involving different European organisations. The target GMES system therefore inevitably involves highly heterogeneous networked systems. In emergency situations, further, the context is also necessarily highly dynamic in terms of changing systems, mobility, and fluctuating operating conditions. This area therefore provides a strong example of the need for on-the-fly solutions to interoperability.

We concentrate on the particular case of a networked system connecting with a video capturing networked system. Various concrete system are able to

[1] `http://www.gmes.info/`

capture video: fixed cameras, robots with video sensing capabilities (UGV: Unmanned Ground Vehicle), or flying drones (UAV: Unmanned Aerial Vehicle). In addition, the videos may be accessed from other heterogeneous systems, including applications run on the mobile handheld devices of the various actors on site, and the ones executed by Command and Control—C2—centres (see Figure 1).

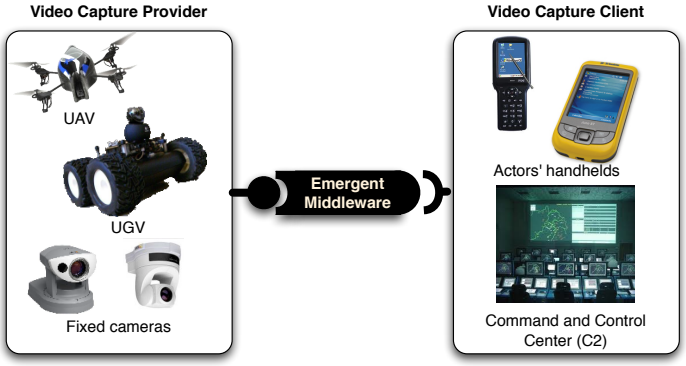


Fig. 1 The GMES case study

Specifically, we focus on two networked systems from Figure 1: C2 and UGV. The C2 system needs to gather information from different cameras in order to analyse them and then makes decision about the appropriate emergency procedure to take. C2 has been developed to interact with a fixed camera and retrieve the videos for given periods of time; for this it uses the SOAP RPC protocol[2]. In contrast, the UGV system captures video and displays it using HTTP Live Streaming (HLS)[3]. It can also move according to some pre-defined patterns, zoom, or get a URL where the video capture can be viewed.

2.3 The Case for Emergent Middleware

Designing middleware protocols for prescribed contexts is not sufficient, rather we support the vision of emergent middleware [26,23] as first introduced in Section 1. That is, where two networked systems are willing to interoperate and are mutually compatible in terms of the required and provided functionality then the middleware software to coordinate the exchange is synthesised (taking into account the respective operating context and environmental conditions of the two systems). Due to the highly heterogeneous and spontaneous nature of potential interactions, the engineering of Emergent Middleware is significantly different from traditional statically developed middleware products.

[2] `http://www.w3.org/TR/soap/`
[3] `http://tools.ietf.org/html/draft-pantos-http-live-streaming`

The approach to achieve such emergent software is based upon the following key requirements:

- *The creation and maintenance of runtime models of individual networked systems* (See Figure 2-❶). In order to reason about how to interoperate with a given system we need to create a runtime model of its application interface and also the middleware protocols that implement this interface such that it can be used remotely. The behaviour of the system must also be modelled in terms of the sequence of operations that are necessary to achieve a particular service. Importantly, to underpin runtime solutions to interoperability these models must capture meaning [7]; that is, given the two networked system models it must be possible for an interoperability solution to understand and reason where systems are semantically similar. For this purpose, we require the use of ontologies as a further extension to the model@runtime, i.e. the elements of the runtime model reference concepts defined in a domain-specific ontology (See Figure 2-❺)..
- *Monitoring and discovery of existing networked systems* (See Figure 2-❷). In order to build a runtime model, the operation of the networked system must be first discovered and then monitored. This requires the extraction of information about the systems using traditional resource and service discovery protocols, e.g., lookup facilities as provided by protocols such as Service Location Protocol (SLP), or Web Services Service Discovery (WS-Discovery), descriptions using languages such as Web Services Description Language (WSDL), and approaches promoting the use of ontology-based techniques to semantically match requests and advertisements [24,31].
- *Learning of networked system behaviour* (See Figure 2-❸). Using the initial discovered information as a starting point, machine learning approaches are required to learn how one must interoperate with a particular system in order to achieve particular behaviour, i.e., this will inform how to model exactly the behaviour of the networked system in terms of its middleware and application protocols.
- *Synthesis of interoperability software* (See Figure 2-❹). We require synthesis solutions that can use the runtime models of two systems to calculate a mediator that will resolve the differences between the heterogeneous protocol endpoints, and then generate the software that implements this mediator on the fly in order for it to dynamically deployed between two systems [25]. The synthesis is further supported by ontologies that formalise the domain knowledge [7].

2.4 Summary

We have advocated the need for emergent middleware and identified the key requirements towards achieving this goal: namely, how discovery, learning and monitoring are required to build the initial models of networked systems; then these runtime models can be reasoned about to synthesise the software

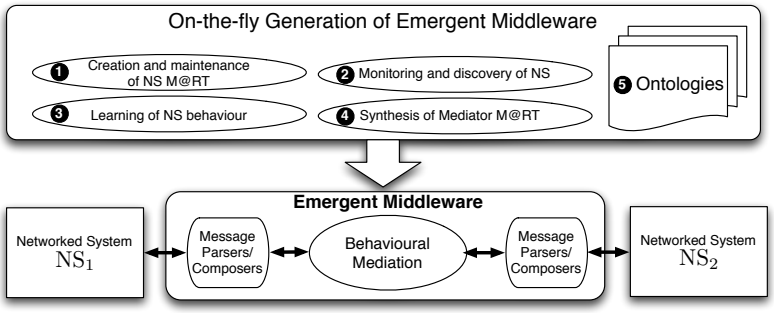


Fig. 2 Supporting Emergent Middleware

artefacts that physically resolve the interoperability problems. This is represented in Figure 2. We now consider in the remainder of the paper how models@runtime naturally meet these requirements, and provide a novel solution that goes significantly beyond what is possible with design-time approaches.

3 Models@Runtime in Action: Sustaining the Dynamic Synthesis of Emergent Middleware

3.1 Overview of the Approach

Figure 3 outlines our overall approach for synthesising emergent middleware between two networked systems NS_1 and NS_2. The key philosophy of this approach is to utilise runtime models to both i) support the reasoning about what emergent middleware should be created, and ii) support the creation of this software artefact itself. Our approach is incremental as illustrated by the following steps seen in the figure:

1. Inference of the runtime models abstracting different concerns of the networked systems, that is *reification*. Reifying the knowledge that has been discovered and learned means that the knowledge has been formulated and made available for computational treatment in the form of a runtime model.
2. Analysis of the runtime models and generation, in the appropriate cases, of the necessary mediator model that will allows the networked systems to interoperate. This implies performing the necessary translation between semantically equivalent operations and coordinating their behaviour, what we call *synthesis of the mediator model*.
3. Concretisation of the mediator model in an artefact (i.e., the Emergent Middleware) that further deals with message-level interoperability, that is *deployment*.

The runtime models in figure 3 capture both the functional and behavioural semantics of networked systems. The functional semantics describe the func-

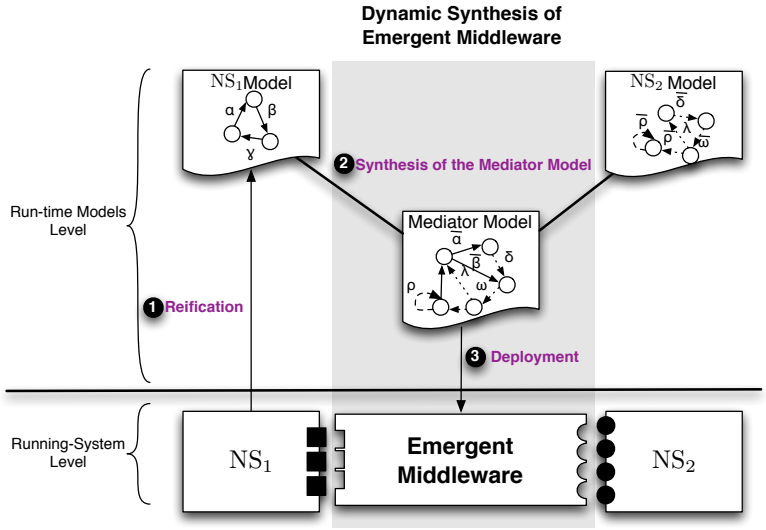


Fig. 3 Overview of the Approach

tionality of the system (i.e., what the system does), the interface actions it requires and/or provides, and the specific middleware platform it is implemented upon. The behavioural semantics specify how the actions of its interface are coordinated in order to achieve the system functionality (i.e. how the system interacts with its environment).

Although the specification of system functionality and behaviour has been acknowledged as crucial in open environments [2], most legacy applications only display their 'syntactic' interface description. Hence, only partial knowledge about the system can be discovered. To infer additional knowledge about the functional and the behavioural semantics of each networked system, we rely on statistical learning to extract their functionality [5] and possibly the ontology-based annotations of their interface [35] and on automata learning to compute their behaviour [30]. Hence, reification (see Figure 3-❶) amounts to the two phases of discovery and learning.

Our aim is to enforce interoperation between functionally-compatible systems, i.e., those requiring/providing semantically compatible functionalities, despite disparities in their interfaces and/or behaviours, by synthesising the *mediator model* that addresses these mismatches and guarantees their behavioural compatibility (see Figure 3-❷).

The mediator model is refined into a software artefact that further takes into account the communication protocols characteristics and is deployed as an Emergent Middleware (see Figure 3-❸).

We rely on ontologies to describe the functional semantics of networked systems from application down to middleware and network layers. Ontologies capture the domain model, which includes the concepts, the functions, and the

data related [1]. Besides formalising and standardising the domain terminology,we use ontologies to describe the underlying middleware and how actions from the application layer are related to this middleware. It also serves describing network-level messages and mapping them by reasoning on their semantics. It is particularly important to highlight the role of ontology reasoning that allows us to infer the relations between concepts in open environments [7], i.e., environments that consists of many interacting systems that are developed by different venders and are either absolutely unaware of or have only partial knowledge about the global system.

3.2 The Runtime Model Specification

The runtime models are the central elements that allow us to reason about how to make systems interoperable. These models need to specify adequate knowledge about the networked systems from the application down to the network layers, as well as the domain knowledge using ontologies. In our approach, two distinct types of models are used to describe the different constituents of the system: the *networked system model* and the *mediator model*. These models are manipulated (reified, transformed, refined) so as to manage the full cycle of interoperability assurance.

The Networked System Model abstracts the interface together with the functional and behavioural semantics of a networked system. For example, and as depicted in Figure 4, the C2 requires a video capturing functionality while the UGV provides it.

Further, we rely on ontologies to describe the interfaces of networked systems. The *interface* defines the set of observable *actions* that the system requires/provides from its running environment. An *input action* $op(in)$:*out* (where *op*, *in*, and *out* belongs to the same domain ontology) requires an operation *op* for which it produces some input data *in* and consumes the output data *out* . For example, the C2 interface includes the *getMPEGVideo(Camera, Period) : MPEGVideo* action that specifies that C2 provides a *Camera* and a *Period* objects and expects an MPEG video in return. The dual *output action*[4] $\overline{op}(in)$: *out* uses the inputs and produces the corresponding outputs. For example, the UGV interface includes the *getVideoRTPAddress(Camera, Period) : VideoAddress* action that expects *Camera* and *Period* objects and returns an address to a video. In addition, the binding defines the specific middleware used in the implementation of a networked system to realise these actions. More specifically, a domain-specific language called the Message Description Language (MDL) [8] specifies the structure and content of network-level messages exchanged to perform each interface action. For example, the C2 binding is implemented using SOAP-RPC, which involves exchanging XML-based messages according to the request/response pattern.

[4] Note the use of an overline as a convenient shorthand to denote output actions

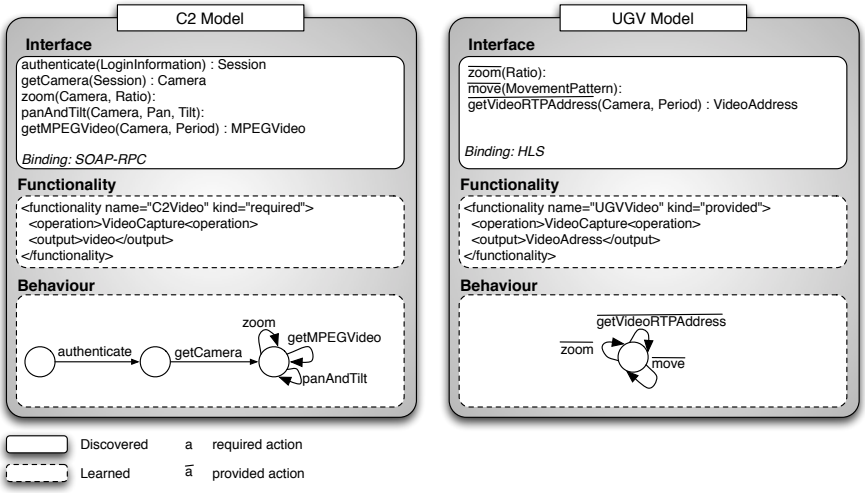


Fig. 4 The models of individual networked systems

These annotated actions are used to define the Labeled Transition Systems (LTS) [28] that represent the behaviour of networked systems. LTSs proved to be effective for describing, understanding and reasoning about concurrent systems. An LTS is a directed graph with labels on each edge describing the progress of the system behaviour when the action, to which the label is attached, is performed. Hence, most of the focus in an LTS is given to the relation and sequencing of actions and not to the meaning of the actions themselves that is also relevant in our case. For example, the C2 system first performs the authentication action, it selects a camera, then it can zoom, send a pan/tilt command to the camera, or ask for a video concerning a given period of time.

The Mediator Model abstracts the necessary translations and coordination to make heterogeneous networked systems interoperable. It includes the models of the involved NSs, a set of pre-defined translation functions used to perform the necessary transformations between semantically equivalent actions that have different syntactic representations. It also embeds a set of pre-defined bindings so as to allow the realisation of actions of the networked system interface. Finally, the behaviour of the mediator model is synthesised in order to allow the successful coordination of the networked systems, i.e., the global system composed of the two networked systems and the mediator is free from deadlock and unspecified receptions. The synthesis of this behavioural model is described in the following section.

The mediator between C2 and UGV (see Figure 5) coordinates the actions of each system and makes the necessary translations. It ensures the translation of semantically equivalent actions using the available transformations functions, which can also be Extensible Stylesheet Language Transfor-

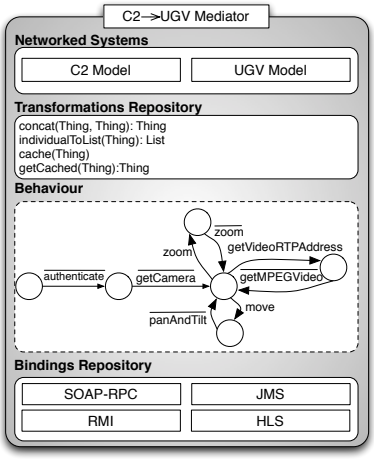


Fig. 5 The models of the C2 → UGV mediator

mations[5](XSLT), e.g., getting the MPEG video required by the C2 from the HLS stream provided by the UGV.

3.3 Building Runtime Models

We have previously outlined the specification of the runtime models; here we describe how both of these models are created using only runtime available information.

Reifying the networked system model. Prior to any spontaneous interaction, we need to sense systems in the environment. This is the role of *discovery*. In particular, we build on state-of-the-art interoperable discovery methods [10, 12,19] to cope with the heterogeneous discovery protocols that exist. Specifically, a *Discovery Enabler* listens on various multicast addresses used by legacy discovery protocols (e.g., Service Location Protocol[6], WS-Discovery[7], UPnP-SSDP[8], and Jini[9]); it intercepts both the advertisement messages and lookup request messages that are sent within the network environment and processes them using appropriate plug-ins. However, only partial models are provided by legacy applications—in most of the cases, only the syntactic interface. Consequently, we rely on learning techniques to complete the model of the networked systems.

[5] http://www.w3.org/TR/xslt

[6] http://www.openslp.org/

[7] http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf

[8] http://www.upnp.org/

[9] http://www.jini.org/

A *Learning Enabler* dynamically infers the functionality and the ontology-based annotations of the interface, and determines the behaviour of a networked system given the discovered interface. The learning of systems' functionality is performed using statistical learning in two steps. First, at design time, examples of interfaces and the corresponding functionality are used to train the algorithm in order to extract a *categorisation function*. More specifically, the algorithm uses natural language text to infer the likelihood between some words occurring in the interface specification and the functionality of the system. Second, at runtime, the interface is analysed in order to infer the appropriate functionality. The accuracy of such inference depends on the size of the examples used during the training phase [5].

Behavioural learning is an iterative process by which a hypothesis behavioural model of the system is incrementally refined according to test-based interaction with the corresponding system. Consider for example the C2 system, after the discovery of the system interface at t_0 (se Figure 6), the first step of the learning process (at t_1) consists on assuming one single state where all the actions can be performed. However, when trying to interact with the system by performing for example a *zoom* then *authenticate*, an error (or exception) might be raised then the model is subsequently refined (t_2). Similarly, when first authenticating then zooming, an error may occur, so the system continue to be refined as in t_3. However, in the general case, such an algorithm guarantees neither the completeness nor the completeness of the learned model [43], which means that this model may continue to change or evolve. This is crucial for the evolution of runtime models explained is Section 3.5.

Synthesising the mediator model. Networked systems are interoperable if they are both *functionally* and *behaviourally compatible*. They are functionally compatible if the provided functionality subsumes the required one, i.e., at a high enough level of abstraction, the functionality provided by one application is semantically equivalent to that required by the other. They are behaviourally compatible if the composed system is free from deadlocks and unspecified receptions [49].

Nevertheless, existing formal notions to behavioural compatibility [38] assume close-world settings, i.e., the use of the same actions to define the behaviour of the systems. What is needed is a notion of compatibility that further takes into account the semantic compatibility of actions and relies on an intermediary system, i.e., the Emergent Middleware, to address their syntactic differences. Towards this goal, we infer the correspondence between the actions of the systems' interfaces so as to generate the mapping processes that perform the necessary translations between actions. Various mappings relations may be defined. They primarily differ according to their complexity and conversely proportional flexibility. These mappings are generated according to the mediator capabilities, which includes receiving and sending messages, delaying the delivery of messages, and reasoning about the semantics of actions in order to generate actions by transforming and composing the original ones. The mediator cannot create except for basic control, such as simple acknowledgments.

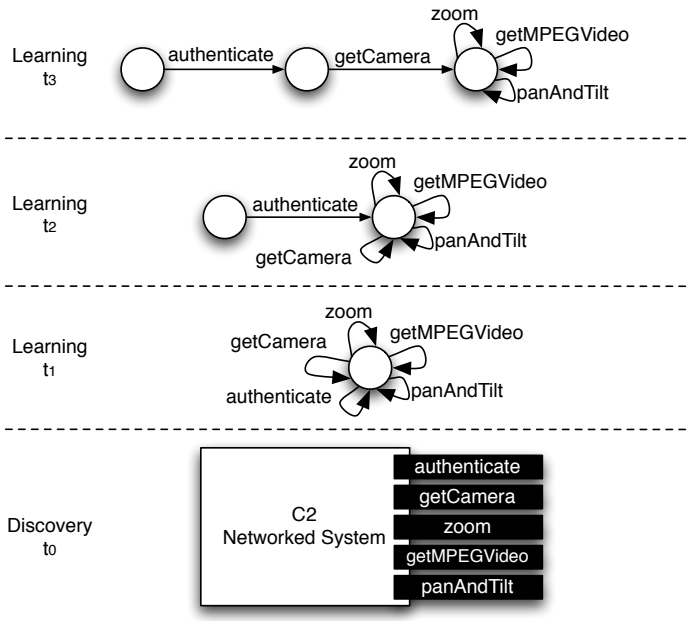


Fig. 6 Learning the behaviour of the C2 networked systems

The interface mapping is performed at runtime and does not require *a priori* knowledge about the systems; the description of these interfaces only need to adhere to the same ontology, which is defined to reflect the shared understanding of the application domain. To synthesise the mediator, we use an ontology-based model checking technique to explore the various possible mappings in order to produce a correct-by-construction mediator that guarantees that the two systems can successfully interact. Model checking is an appealing technique to assess system correctness and automatically verify concurrent systems by exhaustively exploring the state space. While the complete coverage of this state space may lead to state explosion, many solutions have been proposed to alleviate this issue and lead to interesting results when applied at runtime [11].

3.4 Leveraging Runtime Models

In order to produce a concrete mediator we need to leverage the runtime models directly, because the only information about the requirements of the mediator are obtained at runtime. Here, a concrete mediator is a software artefact that can be deployed in a communication network to bind two networked systems together in order that they interoperate with one another.

Interpretation is the foundation of concrete mediators; that is, a domain specific interpreter that executes the actions of an LTS is utilised. This is il-

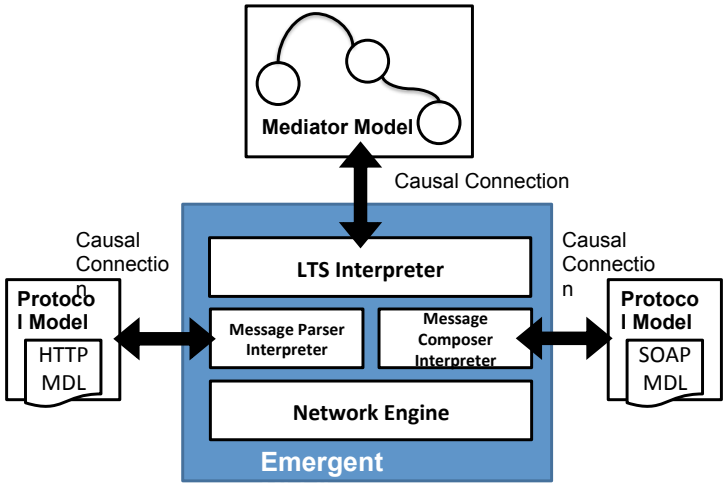


Fig. 7 Runtime models building a connector

lustrated in Figure 7; here an instance of the Starlink framework [8] forms a running connector. However, this is an abstract element until it has been specialised with runtime models to describe the required behaviour. The mediator model is interpreted by an LTS interpreter within the Starlink instance. Each transition action is interpreted to extract the information required to perform a physical middleware-based interaction. The *operation* label, the *input* parameters, and the *output* parameters form the content of the interaction. The *primitive* then informs the interpreter of the middleware protocol to use to physically execute this action content. For example, this could be an RPC invocation using the SOAP protocol.

Importantly, the communication protocols themselves are also modelled; their messages are described in a message description language (MDL) such that their packets can be dynamically composed and parsed based upon this description. A model of each protocol is again plugged into the message composer and parsers interpreters. Hence, when Starlink executes a transition of the mediator's LTS it can produce the correct concrete message. Note, concrete messages are communicated via the network engine (which provides a simple subset of network transport behaviour to physically connect the two legacy systems).

The model of a mediator (as exemplified in Figure 5) therefore consists of sufficient concrete information to execute a mediator. For example, if the two systems are implemented as follows: the C2 is system implemented using SOAP and the UGV system using HTTP-based HLS then in order to perform the `zoom` operation (which is provided by both systems) the mediator must form a SOAP service side of the action to interact with the C2 client and a HTTP client side of the action to interact with the UGV service.

The benefits of interpretation are that a causal connection is inherent in the deployment. The runtime model informs the mediator behaviour, and hence any changes made to the model on-the-fly are automatically and transparently applied. This is similarly achieved at the middleware level; if the mediator migrates to a different communication protocol, or the protocol itself changes (e.g. a version change) then only the middleware model needs to be changed.

3.5 Evolution of Runtime Models

The system needs to evolve as new knowledge is being discovered or learned and to reflect changes in the operating environment. Therefore, the system is monitored continuously to identify executions that do not conform to the learned behaviour of networked system. This verification is carried out at runtime and the model of the mediator is updated according to the changes in the networked system model. Due to the inherent causal connection, the emergent middleware is adapted accordingly so as to reflect the changes to the mediator model. Ontologies may also evolve over time [34], although less frequently. In addition, as ontologies keep emerging and getting standardised, a critical issue is then matching ontologies. The logic grounding of ontologies enables a more accurate matching of concepts compared to syntactic based techniques. Nevertheless, this matching is given with a certain confidence that is never absolutely precise. Quantitative analysis and probabilistic model checking may reveal very useful when quantifying expected behaviour or the confidence on the learned concept and to adapt the overall system accordingly [11, 21].

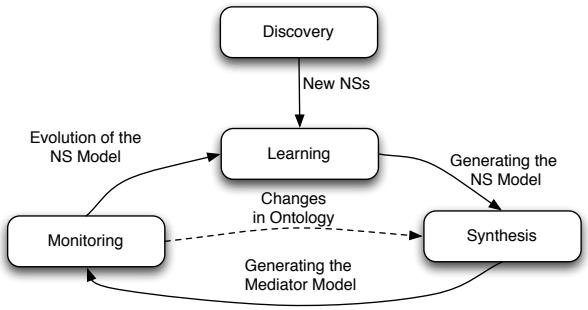


Fig. 8 Evolution of Runtime Models

As depicted in Figure 8, the system is in this case a closed-loop system to better deal with the partial knowledge it has about the environment. Indeed, closed-loop systems have been recognised as fundamental to deal with uncertainty [2, 21]. Then a major challenge is to manage efficiently the changes of the networked systems models in order to re-synthesise the mediator in an incremental way.

4 Related Work

The related work is presented in three different domains: Self-representation and Reflection, Mediation at Runtime and Software Synthesis at Runtime. We also talk about future trends in the area.

Mediation at Runtime. Mediation is an approach that allows the composition of heterogeneous systems by introducing an intermediary software entity to perform the necessary translation and co-ordination while keeping them loosely coupled [47]. It has deserved a large amount of work in various domains, e.g., database integration [27], communication-protocol conversion [29], component adaptation [49], control-system supervision [44], connector wrapping [42], and Semantic Web Services composition [45]. While the generation of these mediators was mainly a design-time concern, the increasing openness of today's highly dynamic and complex systems make the mediator generation shifts towards runtime. Denaro *et al.* [17] propose a solution to interoperability across different implementations of the same standard interface in two phases. At design time, developers define common misuse cases of the interface and define the corresponding healing strategies. At runtime, the system is tested against these case and the appropriate healing strategy is applied. At the architectural-level, Chang *et al.* [14] propose to use *healing connectors* to solve integration problem at runtime by applying healing strategies defined by the developers of the Commercial off-the-shelf (COTS) components.

In order to deal with the open and unconstrained nature of today's complex systems, WSMO [15] defines a formal description language that integrates ontologies with state machines to represent Web Services. It also proposes a runtime framework, the Web Service Execution Environment (WSMX), to mediate interaction between heterogeneous services by inspecting their individual protocols and performing the necessary translation on basis of pre-defined mediation patterns. However, there is no guarantee that the composition of these patterns will not lead to a deadlock. To ensure the correctness of mediation, Cavallaro *et al.* [13] consider the semantics of data and relies on model checking to automatically identify mapping scripts between interaction protocols. Nevertheless, they propose to perform the interface mapping beforehand so as to align the actions of both systems. However many mappings may exist and should be considered during the mediator generation. Indeed, the interface and behavioural descriptions are inter-related and should be considered in conjunction. Moreover, they focus on the mediation at the application layer assuming the use of Web services for the underlying middleware. Starlink [9] proposes a domain-specific language to describe interaction protocols—at both application and middleware layers—to be made interoperable as well as the translation logic to compose them. While these descriptions are made at design time, their interpretation happens at runtime.

The aforementioned research initiatives have made excellent contributions. However, in environments where there is little or no knowledge about the systems that are going to meet and interact, the generation of suitable mediators must happen at runtime whereas in all these approaches, the mediator models

or some mediation strategies and patterns are known *a priori* and applied at runtime. In our approach, the construction of the individual models as well as the generation of the appropriate mediator are performed at runtime

Self-representation and Reflection. Several research approaches have used architectural-based models as the runtime representations of the system to support the treatment of runtime phenomena [48,16,18,32]. In contrast to runtime models that represent directly behavior as in our work, architectural runtime models represent structural views of the running system. Different from [48,16,18], and as in [32], we maintain runtime models causally connected with the running system (i.e. we use reflection). Finally, different from those approaches that use architectural runtime models to represent structural views of the running system, our approach deals directly with behaviour based on the LTS-based models and not just with architectural notions.

Software Synthesis at Runtime. Morin *et al* [32] and Welsh *et al.* [46] also synthesize software artifacts supported by the runtime models. The authors of [32] generate the adaptation logic (i.e. reconfiguration scripts) to reconfigure the system by comparing the current configuration of the running system with the model which represents the target configuration. In [46], Welsh *et al.* also generate the adaptation logic, but different from [32], they use runtime requirement models. None of them inferred information from the runtime system to create the runtime models, i.e. in their cases the runtime models structure is defined before execution. In our case we have used machine learning techniques to infer knowledge that will be used to create the runtime model.

Future trends in Models@run.time. As seen above and in [6,4], models@run.time so far has been used in different areas e.g. dynamic architectures, self-adaptation, and requirements-aware systems [39] among others. These research initiatives have focused on runtime models which specification is designed before the system's execution. However, recently, researchers have started to devise the use of runtime models in providing intelligent support to software at runtime [3] as the line between development models and runtime models gets blur [6]. Also, runtime models look useful when tackling the uncertainty [46] common in the modern and future software systems [21]. To be able to design these future software systems, inferring the knowledge necessary to conceived runtime models during execution is crucial. In this paper, we have shown early results to conceived runtime models, based on information about the running system and inferred using learning techniques during runtime. Authors of [41] have also worked on this topic. Finally, we highlight the need of efficient formal methods at runtime to provide effective support in producing high-integrity systems [11].

5 Conclusions

In this paper we have presented a novel approach to use runtime models to support the dynamic synthesis of software. We have shown how the required

runtime models are automatically inferred and refined by exploiting learning and synthesis techniques, and used at runtime to achieve dynamic interoperability. The approach exploits ontologies to facilitate the mutual understanding and performing the matching and mapping between the networked systems services that need to interoperate.

The dynamic software synthesis approach relies on a formal foundation. During runtime, mediators are formally characterised to allow the runtime synthesis of software. To do that, LTS based models are used to define the matching and mapping relationships between mismatching protocols. These relationships allow the formal definition of the algorithm that synthesises the mediator.

Additionally to the above, in contrast to the use of models in a traditional way, the runtime models presented support reasoning about information that was not necessarily known before execution.

We argue that to support future-proof software systems, the focus of software development should shift from a traditional approach where environmental conditions are foreseen and behaviour of the system is coded accordingly to a more dynamic approach. With the new approaches, components and/or services are dynamically discovered and then composed together to recreate the system required according to the current requirements and environmental contexts. This dynamic composition requires the synthesis of software on-the fly as shown in this paper. We believe that the use of runtime models will play an important role and hope that the approach we have presented provides the foundations for further advances in the area.

Acknowledgements This work is carried out as part of the European FP7 ICT FET CONNECT (`http://connect-forever.eu/`) project. The authors would like to thank Antoine Leger from Thales for valuable input about the GMES case study.

References

1. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
2. Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 2006.
3. Luciano Baresi and Carlo Ghezzi. The disappearing boundary between development-time and run-time. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010.
4. Nelly Bencomo, Gordon S. Blair, Franck Fleurey, and Cédric Jeanneret. Summary of the 5th international workshop on models@run.time. In *MoDELS Workshops*, pages 204–208, 2010.
5. Amel Bennaceur, Johansson Richard, Moschitti Alessandro, Spalazzese Romina, Daniel Sykes, Rachid Saadi, and Valérie Issarny. Inferring Affordances Using Learning Techniques. In *International Workshop on Eternal Systems (EternalS'11)*, 2011.
6. Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *Computer*, 42(10):22–27, 2009.
7. Gordon S. Blair, Amel Bennaceur, Nikolaos Georgantas, Paul Grace, Valérie Issarny, Vatsala Nundloll, and Massimo Paolucci. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In *Middleware'11*, 2011.

8. Yérom-David Bromberg, Paul Grace, and Laurent Réveillère. Starlink: Runtime interoperability between heterogeneous middleware protocols. In *Proc. ICDCS*, 2011.
9. Yérom-David Bromberg, Paul Grace, Laurent Réveillère, and Gordon S. Blair. Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In *Middleware*, pages 390–409, 2011.
10. Yérom-David Bromberg and Valérie Issarny. Indiss: Interoperable discovery system for networked services. In *Middleware*, pages 164–183, 2005.
11. Radu Calinescu and Shinji Kikuchi. Formal methods @ runtime. In *Monterey Workshop*, pages 122–135, 2010.
12. Mauro Caporuscio, Pierre-Guillaume Raverdy, Hassine Moungla, and Valérie Issarny. ubisoap: A service oriented middleware for seamless networking. In *ICSOC*, pages 195–209, 2008.
13. Luca Cavallaro, Elisabetta Di Nitto, and Matteo Pradella. An automatic approach to enable replacement of conversational services. In *ICSOC/ServiceWave*, 2009.
14. Hervé Chang, Leonardo Mariani, and Mauro Pezzè. In-field healing of integration problems with cots components. In *ICSE*, pages 166–176, 2009.
15. Emilia Cimpian and Adrian Mocan. WSMX process mediation based on choreographies. In *Proceedings of Business Process Management Workshop*, 2005.
16. Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 21–26, New York, NY, USA, 2002. ACM.
17. Giovanni Denaro, Mauro Pezzè, and Davide Tosi. Ensuring interoperable service-oriented systems through engineered self-healing. In *ESEC/SIGSOFT FSE*, pages 253–262, 2009.
18. Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.
19. Carlos A. Flores-Cortés, Paul Grace, and Gordon S. Blair. Sedim: A middleware framework for interoperable service discovery in heterogeneous networks. *TAAS*, 6(1):6, 2011.
20. Robert B. France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE*, pages 37–54, 2007.
21. David Garlan. Software engineering in an uncertain world. In *FoSER*, pages 125–128, 2010.
22. J.C. Georgas, A. van der Hoek, and R.N. Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer*, 42(10):52 –60, oct. 2009.
23. Paul Grace, Gordon S. Blair, and Valérie Issarny. Emergent middleware. *ERCIM News*, 2012(88), 2012.
24. Armin Haller, Emilia Cimpian, Adrian Mocan, Eyal Oren, and Christoph Bussler. Wsmx - a semantic service-oriented architecture. In *ICWS*, pages 321–328, 2005.
25. Valérie Issarny, Amel Bennaceur, and Yerom-David Bromberg. Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In *SFM-11*, volume 6659 of *Lecture Notes in Computer Science*. Springer, 2011.
26. Valérie Issarny, Bernhard Steffen, Bengt Jonsson, Gordon S. Blair, Paul Grace, Marta Z. Kwiatkowska, Radu Calinescu, Paola Inverardi, Massimo Tivoli, Antonia Bertolino, and Antonino Sabetta. Connect challenges: Towards emergent connectors for eternal networked systems. In *ICECCS*, pages 154–161, 2009.
27. Vanja Josifovski and Tore Risch. Integrating heterogenous overlapping databases through object-oriented transformations. In *VLDB*, pages 435–446, 1999.
28. Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 1976.
29. Simon S. Lam. Protocol conversion. *IEEE Transaction Software Engineering*, 1988.
30. Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation learnlib. In *TACAS*, pages 220–223, 2011.
31. Sonia Ben Mokhtar, Pierre-Guillaume Raverdy, Aitor Urbieta, and Roberto Speicys Cardoso. Interoperable semantic and syntactic service discovery for ambient computing environments. *IJACI*, 2(4):13–32, 2010.
32. Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models at runtime to support dynamic adaptation. *IEEE Computer*, pages 46–53, October 2009.

33. Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE*, pages 122–132, 2009.
34. Natalya Fridman Noy, Abhita Chugh, William Liu, and Mark A. Musen. A framework for ontology evolution in collaborative environments. In *International Semantic Web Conference*, pages 544–558, 2006.
35. Nicole Oldham, Christopher Thomas, Amit P. Sheth, and Kunal Verma. METEOR-S web service annotation framework with machine learning classification. In *SWSWPC*, pages 137–146, 2004.
36. Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.
37. Peyman Oreizy, David S. Rosenblum, and Richard N. Taylor. On the role of connectors in modeling and implementing software architectures. Technical Report 98-04, University of California, Irvine, 1998.
38. Meriem Ouederni and Gwen Salaün. Tau be or not tau be? - a perspective on service compatibility and substitutability. In *WCSI*, pages 57–70, 2010.
39. Pete Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. *Requirements Engineering, IEEE International Conference on*, 0:95–103, 2010.
40. Douglas C. Schmidt. Model driven engineering. *IEEE Computer*, pages 25–31, 2006.
41. Hui Song, Gang Huang, Yingfei Xiong, Franck Chauvel, Yanchun Sun, and Hong Mei. Inferring meta-models for runtime system data from the clients of management apis. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part II*, MODELS'10, pages 168–182, Berlin, Heidelberg, 2010. Springer-Verlag.
42. Bridget Spitznagel and David Garlan. A compositional formalization of connector wrappers. In *ICSE*, 2003.
43. Bernhard Steffen, Falk Howar, and Maik Merten. Introduction to active automata learning from a practical perspective. In *SFM*, pages 256–296, 2011.
44. Andreas Tolk and J. Mark Pullen. Using web services and data mediation/storage services to enable command and control to simulation interoperability. In *DS-RT*, pages 27–34, 2005.
45. Roman Vaculín and Katia P. Sycara. Towards automatic mediation of OWL-S process models. In *Proceedings of ICWS*, 2007.
46. Kristopher Welsh, Pete Sawyer, and Nelly Bencomo. Towards requirements aware systems: Run-time resolution of design-time assumptions. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 560–563, 2011.
47. Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 1992.
48. Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *In Proc. 26th International Conference on Software Engineering*, pages 470–479, 2004.
49. Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 1997.

Model-based Management of Service Composition

Yun Ma, Xuanzhe Liu, Gang Huang
Institute of Software, School of Electronics Engineering and Computer Science
Peking University
Beijing, China
{mayun11, liuxzh, huanggang}@sei.pku.edu.cn

Paul Grace
School of Computing and Communications
Lancaster University
Lancaster, UK
p.grace@lancaster.ac.uk

***Abstract*—Promoted by the Service Computing paradigm, service composition has played an important role in modern software development. Currently, available services have covered a wide spectrum of heterogeneity, including SOAP services, RESTful services and other data services. The composite services should continuously serve for a large number of users. The heterogeneity and open dynamic network environment bring grand challenges to the management of service composition. Based upon our previous work on service composition middleware – Starlink, and a runtime system management tool – SM@RT, this paper proposes a model-based approach to service composition management at run time. A runtime model enables casual connections between applications and supporting platforms, provides a global view of a running system, abstracts underlying technical details, and performs automated generation of management code. By constructing the runtime model of Starlink and using the SM@RT tool to generate synchronization between model and running composition, our approach makes the following contributions to service composition management: (1) a more comprehensive view of service composition management; (2) an easy-of-use manner to perform management operations at model level without underlying tedious details; (3) an on-the-fly effect on running system by means of synchronization between the model and composite services. We demonstrate that our approach can tackle the challenge of service composition management by using a case study of a photo sharing composite service application.**

Keywords—Model-based Management; Service Composition; Runtime Model

I. INTRODUCTION

Service composition has played an important role in modern software development[1]. More and more computational functionalities have been published as various kinds of services, including SOAP Web services, RESTful services, RSS/Atom data services, etc. Through composition, more functions can be implemented and different user experiences can be gained compared to a single service. A composite service is usually developed and executed on a chosen service composition middleware which enables interoperability between services[2]. At design time, service composition middleware often provides management interfaces for composite service developers to add QoS-related constraints. These constraints can help deal with errors since individual service properties may be violated in the composition. For instance, a client service with a response time constraint may be composed with a server service. The assembler can design to return an error to the client service when the composition fails to get a response from the server service in the given period.

When the composite service application is running, management brings more challenging issues[3]. For one thing, it should monitor the composite service's running status, so that developers can find bugs or bottle necks influencing the quality of service. Some advanced monitoring tasks may include checking the satisfaction of specific rules and generating statistics of running information. For another thing, unexpected errors may occur to break down the execution of composite services, such as: time out, network unreachable or unexpected incoming requests. Basic control actions like: stop, restart, and retry are often provided by service composition middleware to deal with error states while advanced ones allow more detailed configuration according to policies. For example, when a response time out error arises, the composition may first retry sending requests and if it fails for three times then it sends an error message to the other service. Consequently, there is a significant requirement for the management of service composition at runtime.

However, with the rapid adoption of Mobile Computing and Cloud Computing, services are developed for and executed in highly open dynamic network environment and heterogeneous platforms, which makes runtime management of service compositions much more challenging.

On one hand, services become more heterogeneous making compositions more complex. Although web service (SOAP, RESTful API) has been regarded as the de-facto standard in service computing area, more and more functions are published as other kinds of service which are not web services, especially in the pervasive environment. Heterogeneity of these services span largely from network protocol, middleware technology to application processes. For example, TCP or UDP based network transport can be used to meet different data transportation requirements. Message formats (binary, text or XML, etc.), interaction style (RPC, publish-subscribe, data sharing, etc.) and synchronization mode are different features of service execution middleware. Finally, services aiming to achieve the same application goal can behave quite differently, such as the order of requests and number of parameters. Therefore, services in a composition are so heterogeneous that

several related aspects should be taken into account, which increases the management complexity and difficulty.

On the other hand, the rapid evolution of services and dynamically changing environment call for the capability of service compositions to maintain and evolve at run time. Nowadays, in order to meet the users' ever-growing requirements, new services are emerging all the time and existing ones are likely to be updated. This trend leads to the fact that the boundary between service design and execution is vague compared with traditional software development lifecycle. No service assemblers can foresee the emergencies, nor would they allow their composite services stopped, redesigned and restarted. What makes things even worse is that services are being executed in ever-changing environments and the QoS assurance is significantly difficult. For instance, a mobile service user is likely to use a service by Wi-Fi if a wireless hotspot is provided but will change to a 3G network where there is none. The bandwidth, latency and other QoS properties are quite different between these two networks, which results in the change of a high QoS service in Wi-Fi network to a low one in 3G network. The composite services have to identify and adapt to these changes, which also needs to be accomplished at run time.

Addressing the issues above, we try to leverage runtime model for service composition management. A runtime model is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective[4]. It has been broadly adopted in the runtime management of software systems[5][6][7]. With the help of runtime models, administrators can obtain a better understanding of their systems and write model-level programs for management. We have developed a model-based runtime management tool, called SM@RT (Supporting Model AT Run Time[8]), which provides the synchronization engine between a runtime model and its corresponding running system. SM@RT makes any state of the running system reflected to the runtime model, as well as any change to the runtime model applied to the running system in an on-the-fly fashion.

In this paper, we present a model-based approach to service composition management at run time. This approach is based on a service composition middleware – Starlink[9], which was proposed in our previous work and further extended in CONNECT[10] project. Starlink uses automata to describe service behaviour and service compositions from the network layer, the middleware layer to the application layer. Interoperability of services is enabled by interpreting the service composition automata. In our approach, we first construct a meta-model of Starlink service composition and specify the map between this model and Starlink elements by an access model. Then we use the SM@RT tool to generate the synchronization between the runtime model and the running Starlink composite service. Finally, we can utilize model-based techniques or write model-based programs to manage the composite services at runtime.

Our approach makes the following contributions to the management of service composition:

(1) A runtime model provides a comprehensive view correlated with all of the related layers of service composition, from network to middleware protocol and application process. Therefore, administrators are able to monitor the service composition more logically and get better understanding.

(2) Model-based techniques provide a much simpler manner to check constraints and perform control actions to the service composition. Model transformation languages like QVT (Query/View/Transformation, [11]), model constraint languages like OCL (Object Constraint Language, [12]) and model checking methods are efficient tools to program at the abstract model layer. They facilitate administrators to write fewer lines of code, while obtaining much higher management efficiency.

(3) The synchronization engine generated by our SM@RT tool promises that any changes at run time will immediately be reflected in the model view, and any changes to the model will immediately be applied back to the running composite service. With the on-the-fly model checking support, SM@RT can ensure the correctness and consistency of the management at run time.

The rest of this paper is organized as follows: Section II briefly introduces our preliminary work – Starlink and SM@RT; Section III presents a motivating example to illustrate how service composition is hard to manage at run time. In Section IV we describe the overview of our approach, with the details of the runtime model, and how to manage the Starlink service composition with the model at run time. We show how our approach simplifies the service composition management task by a case study in Section V. After comparing our approach with related works in Section VI, we conclude the paper and discuss about future works in Section VII.

II. PRELIMINARIES

A. *Starlink*

Starlink is a service composition middleware, which can create and deploy runtime solutions to interoperability of services using high-level models of service behavior[9]. First, it provides a domain specific language – MDL (Message Description Language) to abstract the middleware message. An AbstractMessage structure is defined to represent different kinds of messages in a unified way so as to simplify the manipulation of messages. Starlink can generate parsers/composers for consistent bi-directional translation between a network-layer message and an AbstractMessages. Second, it uses k-Colored automata to specify both middleware-layer and application-layer service behaviour and gives the correlations between them. Starlink deals with service behaviour as the sending and receiving of messages, which is modeled by transitions of the automata. For the middleware-layer, colour specifies the network related attributes and MDL to abstract the message. For application-layer, colour specifies the binding from application message to middleware message. Third, Starlink represents the service composition by mediator consisting of translation logics, which map the application-layer message from one service to another. Bridge states are

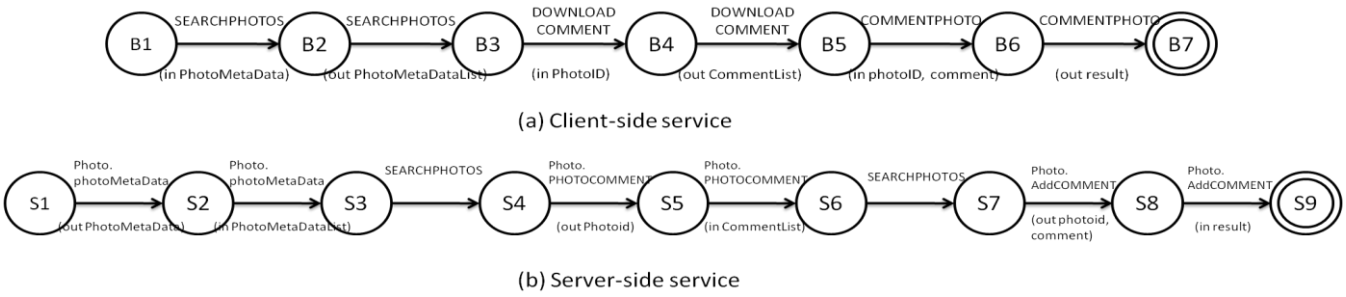


Figure 1 Client-side and Server-side photo sharing services

added to sending and receiving states to allow translation logic (e.g. the translation of data content) to be executed.

Supplied with the automata specification of services to be composed and the mediator, Starlink generates the model of composited service and executes it by interpretation. Starlink has provided us with a very solid ground to resolve management of service compositions since it abstracts all the layers of service heterogeneity by using models.

B. SM@RT

SM@RT (Supporting Model AT Run Time), proposed by our previous work in [8][20][21], consists of a domain-specific modeling language (called SM@RT language) and a code generator (called SM@RT generator) to support model-based runtime system management. The SM@RT language allows developers to specify: (1) the structure of running systems by a UML-compliant meta-model; (2) how to manipulate the system's elements by an access model. With these two models, the SM@RT generator can automatically generate the synchronization engine to reflect the running system to the runtime model. This synchronization engine not only enables any states of the system to be monitored by the runtime model, but also any changes to the runtime model to be applied on the running system. Therefore, developers are able to leverage the existing model-based tools (like OCL, ATL, GMF etc.) to manage running systems.

SM@RT provides a systematic way to manage running systems and we have applied it on several practical systems, including the JOnAS JEE server, which demonstrates very positive results[8]. Here we adopt SM@RT to our service composition system. By building the runtime model of composite services and generating the synchronization engine, administrators can monitor and control the system much more conveniently and comprehensively.

III. MOTIVATING EXAMPLE

In this section, we describe an example to illustrate how Starlink makes services interoperable and to identify the challenges of managing the composition. There are two photo sharing services which allow users to search, download and comment photos. One is client-side, using CORBA[13] as the middleware technology, while the other one is server-side, using XML-RPC[14] protocol based on HTTP. A composite service developer wants to compose the two services to make a photo sharing application. However, he meets with heterogeneity problems.

At the middleware level, CORBA is based on binary messages and uses a very specific protocol (IIOP) to request services, in which the address of service and name of method to invoke are encoded in a hexadecimal representation. XML-RPC is XML-based string messages encoded in the body part of HTTP messages, which uses the HTTP protocol to request services. Therefore, the client-side and server-side services cannot directly communicate with each other.

At the application level, Figure 1 shows the behaviour of the two services by automata on Starlink. For the client-side, it first sends a SEARCHPHOTOS request with parameter PhotoMetaData (like when and where the photo was taken) and gets back the result list. Then it sends the DOWNLOADCOMMENT request with a PhotoID parameter and receives the corresponding CommentList. Finally, a COMMENTPHOTO request with PhotoID and comment content is sent out, and the service is finished. The behaviour of server-side is almost symmetric to the client-side, except that after returning the PhotoMetaDataList and CommentList, it will do some other routines (denoted as SEARCHPHOTOS) before handling the subsequent request.

To compose these two services by Starlink, the developer has to build application-layer behaviour models using k-Coloured automata, write the mediator model and use the IIOP and XML-RPC middleware-layer k-Coloured automata provided by Starlink. Starlink can use these specifications to generate the composition. Figure 2 depicts some fragments of the specifications. 2(a) is the transition from state B2 to B3 of client-side service automaton. 2(b) is the binding from IIOP SEND message to application message. 2(c) is the IIOP behaviour automaton and 2(d) is the MDL of IIOP message. Figure 3(a) shows the merged k-Coloured automaton generated and executed by Starlink. Figure 3(b) is the fragment of its specification. We can see that State B2 and B4 are two bridge states. Each of them is split to two states (B21/B22 and B41/B42) to deal with translation between these two services. Take B4 as an example. It is split to states B41 and B42. The *DOWNLOADCOMMENT* message received in state B41 is translated to *PHOTOCOMMENT* message which will be sent by state S4. The *CommentList* received in state S6 is translated to message compacted with the client-side service and will be sent by state B42.

Obviously the composition design procedure above requires a lot of manual tasks. However, after the composite service is started, the developer may find himself/herself trapped in more tedious management operations.

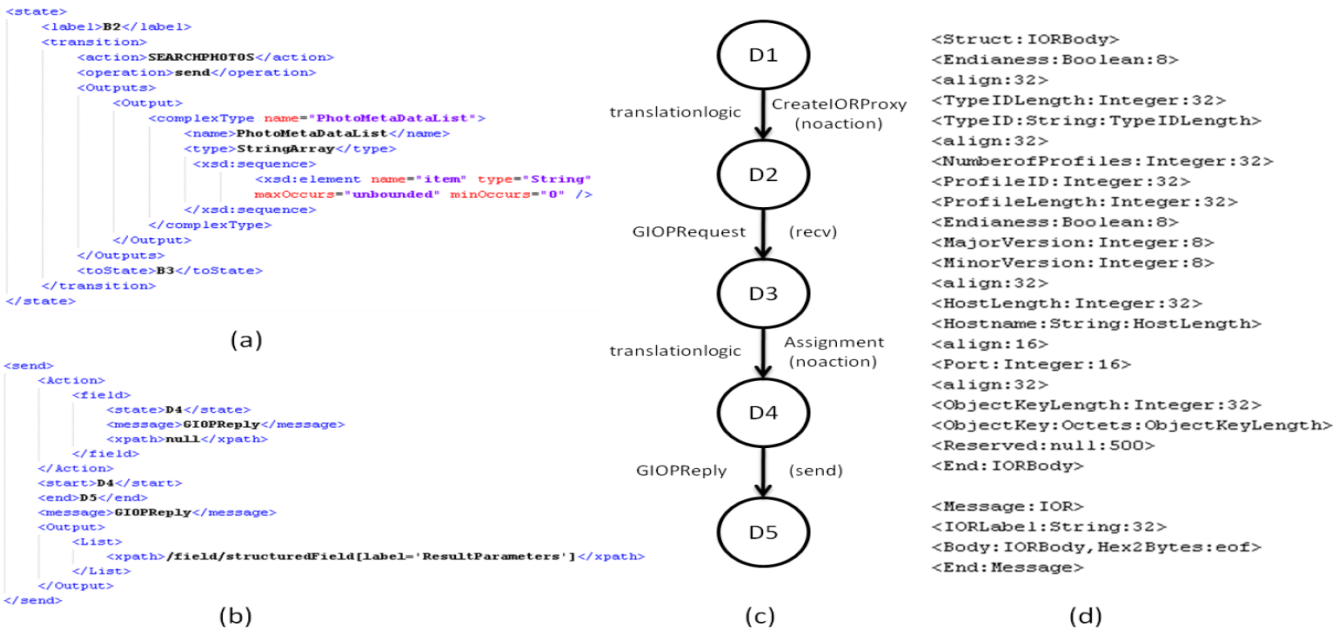


Figure 2 Fragments of the specifications of client-side photo sharing service

First, he/she may need to get information of the running composition, either by referring to a log file or viewing of a GUI interface. This information may include the response time of each request, the number of requests received in a period or even every single request and response message. However, as the composition is related to network layer, middleware layer and application layer, it is very difficult to get a comprehensive view and organize the collected data to meet the developer's satisfaction.

Second, some constraints are likely to be checked to the composite services at runtime, which can be used to ensure functional and non-functional properties of composition. For instance, a constraint of minimum length of comment content can improve quality of the comment list; a constraint that the request *PhotoID* for comment list should be contained in the returned result list in order to avoid unsafe actions accessing the data, etc.

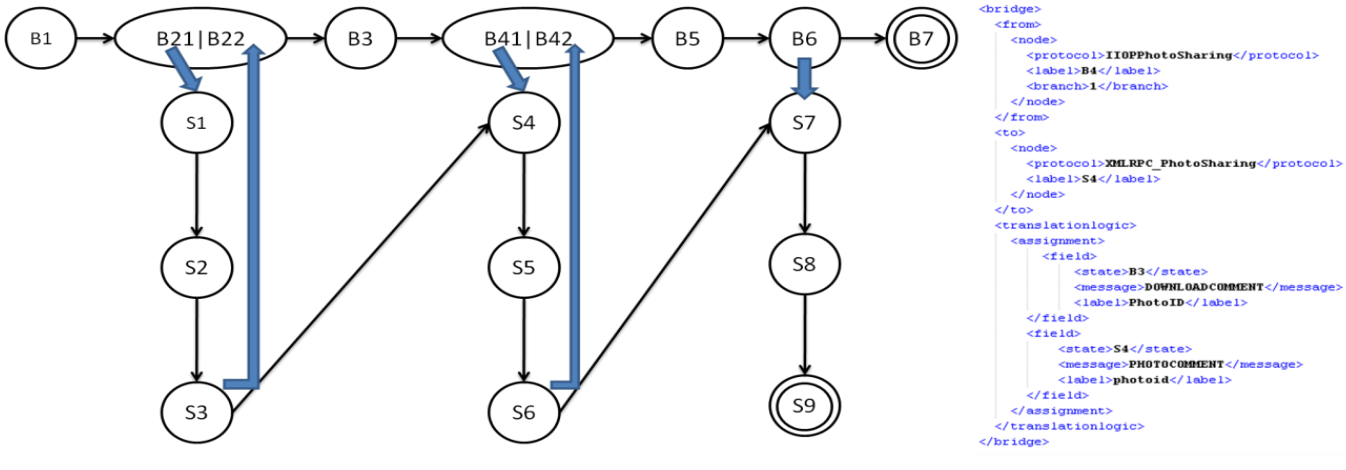
Last but not the least, control actions may need to be performed without stopping or restarting the composition. For example, cuss words are usually found in the comments. The developer may add an additional check-and-replace function that checks the comments returned by server-side service. If there exists words to be censored, they can replace these with alternative characters (e.g.'*'). Another example is that when a timeout error state occurs, the retry policy can be used instead of returning an error immediately. These actions are not always easy to implement especially at runtime since they are likely to require changes of the composite services.

IV. APPROACH

A. *Approach Overview*

The core of our approach is to leverage runtime model for the management of service composition. There are four steps towards our goal to enable runtime management of service compositions (as shown in Figure 4):

(1) Build the system meta-model of a Starlink service composition, by specifying what kinds of elements of such



(a) Photo sharing composition

(b) Fragments of mediator

Figure 3 Photo sharing composition specifications

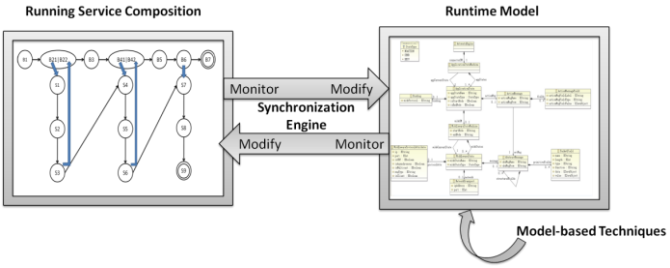


Figure 4 Approach Architecture

compositions can be managed. The meta-model is UML-compliant so that it can be easily understood by developers.

(2) Build the access model of a Starlink service composition middleware, by specifying how to manipulate the manageable elements to monitor and modify them. The access model is a domain specific language based on the system meta-model. Developers can utilize elements in the meta-model and write code fragments to manipulate the system elements, which is also straightforward.

(3) Generate the synchronization engine by SM@RT with the meta-model and access model provided by the first two steps. When the composite service is running, the runtime model (instance of the system meta-model) will reflect the actual state of the composite service.

(4) Utilize model-based techniques (such as QVT and OCL) to monitor, check constraints and perform control actions to the service composition based on the runtime model. The synchronization engine could generate the management code and apply to the composition, thus ease the task of management.

In the next two sections, we will give the detail of each step. Section B presents the Starlink service composition model, including both the system meta-model and the access model. Section C describes how to use the runtime model to address management issues.

B. *Service Composition Model*

As the first step targeting at model-based management, it is essential to have the proper model of service composition. The logic of our approach to construct the model is: (1) we build the meta-model of service composition; (2) we specify the map between this meta-model and the running service composition by an access model; (3) we finally use the SM@RT tool to generate the runtime model, which can reflect the running status of service composition. In this section, we will first present the meta-model of Starlink service compositions and then show the access model. The generated runtime model will be illustrated by a case study in Section V.

1) Meta-model

Figure 5 shows the meta-model of a Starlink service composition. It is clear to divide the model into three layers from bottom to top: network, middleware and application layer, which corresponds to the aspects we discussed in Section I and III that make services extremely heterogeneous. It should be noted that the meta-model in this paper is NOT the total reflection of each detail of the Starlink service composition. The rationale is that we just build what we need for the runtime management tasks. Those elements are omitted which may be important to realizing functions of composition but does not relate to runtime management issues.

(1) **Network Layer.** At the bottom of our model, there is only one class named *NetworkTransport* representing the network information. From the basic view, a service is an endpoint in the network which can send and receive messages. Different services may use different network protocols and even one service can also be bound to different network protocols. Therefore, it is necessary to model the network. Attributes of the *NetworkTransport* class are: *IP* address and *port* of the endpoint; TCP or UDP based transportation denoted by *isUDP*; *isMulticast* represents whether the transport is in multicast mode.

(2) **Middleware Layer**. Nowadays, almost all services are developed and executed on a specific middleware. We have mentioned that Starlink provides a domain specific language – MDL to transform various kinds of message to a unified representation – *AbstractMessage*. Based on this mechanism, Starlink abstracts middleware protocols as k-Coloured

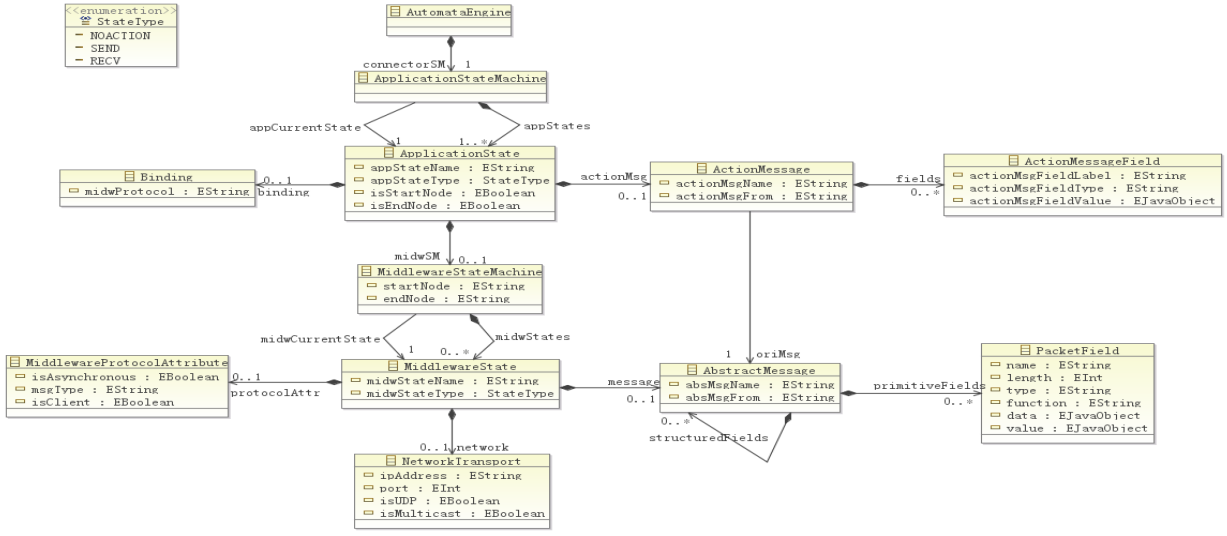


Figure 5 Meta-model of Starlink service composition

automata, in which the colour refers to attributes of the protocol and transitions are message exchanges. In our middleware layer meta-model, we use five classes to describe these abstractions:

- *MiddlewareStateMachine* is the middleware automaton, containing a set of *MiddlewareStates* and a point to the current state. The middleware automaton is a whole description of the middleware protocol including both sending and receiving operations. But for a certain application state, it is either a SEND state or a RECEIVE state. Therefore, we add two string attributes, *startNode* and *endNode* to indicate which part of the middleware automaton is used by the application state.
- *MiddlewareState* is the basic elements of middleware automaton. It has a *name* and *stateType* attribute, representing whether it is a SEND, RECEIVE or NOACTION state.
- *MiddlewareProtocolAttribute* class is combined to each *MiddlewareState*, which is the "colour" of the k-Coloured automaton. It describes the message type (binary, text or XML messages), communication mode (synchronous or asynchronous) and whether it is a server part and client part with respect to RPC style middleware protocol.
- *AbstractMessage* class models the transition of middleware automaton, which is combined to each SEND and RECEIVE *MiddlewareState*. When a certain *AbstractMessage* is received or sent, the current state will transit to another one. This *AbstractMessage* class represents the abstraction of various messages by MDL. Each *AbstractMessage* has a *name* and *from* attribute, composed by a set of *PacketField* and *StructuredField*.
- *PacketField* is the basic element of each message, which has the *name, length, type, function* of the field as well as the raw *data* and the typed *value*. *StructuredField* can be treated a special kind of *AbstractMessage*, which is also composed of a set of *PacketFields*. For instance, an HTTP request message contains *PacketFields* of version, method, URI, host, date etc. and a body Structured fields. An IIOP message has *PacketFields* of typeID, profileID, hostname, port and so on.

Note that we do not model message parsers and listeners at the middleware layer. As you may remember, Starlink can generate parsers/composers to read/write messages from/to network according to the given MDL. However, such information is not needed in management since the middleware message formats are rarely changed and it is not the service administrator's task to handle these changes.

(3) **Application Layer**. On top of the model, it is the application automata representing the business process logic. Starlink enables interoperability of services by generating a connector according to the individual services' k-Coloured automata and the mediator specification. The connector and other related artifacts (like parsers/composers) will be deployed and executed by interpretation. There are six classes in the application layer.

- *AutomataEngine* is the entry point of a Starlink composite service, which provides general control actions like pause stop, restart and so on. It contains a connector, whose type is *ApplicationStateMachine*.
- *ApplicationStateMachine* is the merged application automata, containing a set of ApplicationStates and a point to the current state, just like those in middleware automata. So far, Starlink only supports one automata engine executing one connector. For each connector, Starlink will create a new engine. Therefore, the association relation is one-to-one between *AutomataEngine* class and *ApplicationStateMachine*.
- *ApplicationState* is the basic element of application layer automata. Each state has a *name, type* (SEND, RECEIVE or NOACTION state) and two bool attributes indicating whether it is a start or end state.
- *Binding* class is associated with SEND/RECEIVE states, representing the map from application layer message to middleware layer message.
- *ActionMessage* models the transition of application automata, which has a *name* and *from* attributes. *ActionMessage* has a point to *AbstractMessage*, meaning that each *ActionMessage* is transformed from or to one *AbstractMessage*. The transformation logic is specified by the Binding class. When the middleware automaton receives a message, it will be transformed to an application *ActionMessage* by Binding operations, which subsequently triggers application state transition. When an application state of SEND type triggers transition, the *ActionMessage* will be transformed to *AbstractMessage*, which can then be sent out by executing the middleware automata.
- *ActionMessageField* class is what *ActionMessage* consists of. It has *label, type* and *value* attributes. For example, the client photo sharing application has a SEARCHPHOTOS *ActionMessage* (Figure 2a). It consists of one *ActionMessageField*, whose name is PhotoMetaDataList and type is StringArray. These are useful information with respect to management because administrators can monitor a specific field related to their own requirements and directly add constraints to the field.

2) Access Model

Now we have built the meta-model reflecting the structure of the running composite service. We then specify the map between model and running system in order to generate synchronization engine for manipulation. The access model, named *decmodel*, is provided by SM@RT tool. It contains an *Imports* field for specifying library dependency, a *CommonFeatures* field to write reusable code, and a series of *Map:Class* fields which describe how each class in the model is related to one or more elements of a running system. For each attribute and combination relation of a Class, a

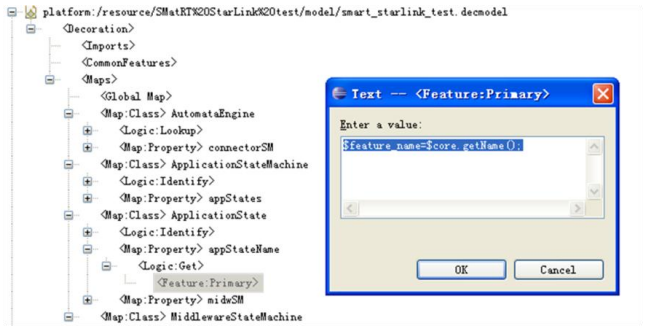


Figure 6 Fragment of access model

Map:Property field should be added to the corresponding *Map:Class*. In *Map:Property*, we can write "get" and "set" code fragments to specify how to get and set values of the attributes.

Starlink provides a set of Java classes that can interpret the models in order to support interoperability solutions. For example, there is a *StateMachine* class to handle k-Coloured automata, an *AbstractState* class to interpret states of automata, a *Bridge* class to interpret bridge states of mediator. According to this, it is simple to map the classes in our meta-model to the Java classes of Starlink. Figure 6 shows a fragment of the access model. In most of the cases, one meta-model class can be mapped to one Java class. The "get"/"set" logic of meta-model class' properties is just to access the corresponding attributes of Java class. However, it is possible that one meta-model class maps to more than one Starlink Java classes, or vice versa. The reason is that the runtime model is used for management and can be different from the system model. For instance, the *startNode* and *endNode* attribute of *MiddlewareStateMachine* is from the *State* Java classes, while the combination relationship with *MiddlewareState* is maintained in *StateMachine* Java classes. Therefore, the meta-model class *MiddlewareStateMachine* has to be mapped to both *State* and *StateMachine* Java classes of Starlink. Another example is the meta-model classes *MiddlewareState* and *MiddlewareProtocolAttribute*. These two classes are mapped to the same Java class *State* because *ProtocolAtrributes* is a very important feature for management and should be separated to handle directly.

C. *Management by Runtime Model*

Based on the meta-model and access model, we use SM@RT tool to generate the synchronization engine between model and Starlink composite service. When the service is running, the fields of meta-model will be assigned with values according to the status of service. Thus we get the runtime model and model-based management can be performed. We can classify service composition management into two categories. One is monitoring, by which an administrator could retrieve needed information of the running services; the other is those runtime changes, by which an administrator is able to change the service's behaviour when unexpected error states occur or add new features to the composite service. Details will be shown in this section on how to achieve management goals using the runtime model.

1) *Monitoring*

Monitoring is the basic but significant management requirement. With monitoring, administrators can capture the runtime status of their service compositions, find and fix errors or bugs in order to improve their composite services. A runtime model not only enables to provide the raw data concerned with details of runtime information, but also to allow advanced monitoring activities including customizing information and checking constraint violations. With the help of the synchronization engine, the runtime model reflects the information of running composite services into the value of its fields. Every time the synchronization is performed, the values will be updated to the up-to-date status. For instance, in the photo sharing example, an administrator can know the number of photos returned by the server-side service and content of every comment sent by client-side service. It is easy to log by means of saving the runtime model onto disk. It makes great difference that the model organizes the information more logically so that administrators can get a comprehensive view of the composite service status.

Moreover, armed with model-based technologies, it is possible to provide advanced monitoring capabilities. Although the runtime model covers almost all aspects of running composite services, it is not the rare case that administrators might prefer to pay attention to only a part of them. For instance, a content supervisor may be only interested in the comments of photos so that only the *ActionMessageField* "comment" of *ActionMessage* "COMMECTPHOTOS" should be taken into account. In our approach, this can be achieved by model transformation techniques. We only need to build another model which only contains related information to the monitoring requirement, and write model transformation rules using QVT. Then the transformation engine can automatically create the customized model and maintain the relationship with the original one. Meanwhile, model checking techniques can be used to check satisfactions of constraints. This reduces administrators work to write low level codes. For example, one constraint is added to photo sharing application that the *PhotoID* sent out to request comment list should be contained in the *PhotoMetaDataList* returned in the previous step. This can be easily checked by writing an OCL expression with the runtime model.

2) *Runtime changes*

As services and network environment are always changing, especially in current Mobile Computing and Cloud Computing paradigm, service compositions should adapt to these changes. Other changes may be due to the improvement of business logic. The runtime model plays an important role in solving runtime changes. Since the synchronization engine maintains the connections between model and composite service, any changes to the runtime model can be immediately applied to the composite service. Runtime changes can be simply handled by either changing the structure of the runtime model, or changing the values of model element fields. For example, if the interface of server-side photo sharing service to get *PhotoComment* by *photoID* is changed, perhaps one parameter is added; in the conventional way, we have to stop the service, write the new composite service and restart it. However, with the help of the runtime model, we just add one more *ActionMessageField* class to the PHOTOCOMMENT

ActionMessage class, and assign it by default value. After synchronization, the composite service will be changed immediately: when the next get photo comment list is requested, it will contain one more parameter with the default value so that this request can be handled correctly by the server-side service. This change is achieved by changing the model structure. An example for changing model field's value is that a limited number of photos can be returned in a free version of photo sharing composite service, while there is no such limitation in the charged version. To realize this value-added function, the administrator only reduces the length of array in *ActionMessageField* "PhotoMetaDataList" corresponding with the SEARCHPHOTOS *ActionMessage* sent to the client-side service. On the whole, both structure changes and field changes are easily performed by model-based programming which enables the developer to write less and achieve more.

V. Implementation and Case Study

A. Implementation

We build the meta-model and access model of Starlink service composition using the Eclipse Modeling Framework [15](EMF), and then generate the synchronization to maintain the causal connections between model and composite service by our SM@RT tool. The tool is built upon EMF. It generates the model listener, model proxy, and system proxy specific to the target system. EMF is an efficient implementation of a core set of MOF. This core set is called EMOF (Essential MOF) in MOF 2.0 and Ecore in EMF. SM@RT generates a Java class for each of MOF classes in the system meta-model, by implementing the EObject interface defined by Ecore. Ecore conforms to UML, since UML is also defined from MOF. Therefore our runtime model is compliant to UML. This makes our model easily understood by developers who are familiar with Object Oriented programming. The runtime model is presented by default in a tree view with editors to set the values of fields so that it is very clear to monitor the running status of composite services. EMF also provides many model-based tools like QVT engine, which can be utilized to perform the model-based management discussed in the previous section. Furthermore, it is very simple to integrate other model manipulation tools into Eclipse, which makes our approach more flexible.

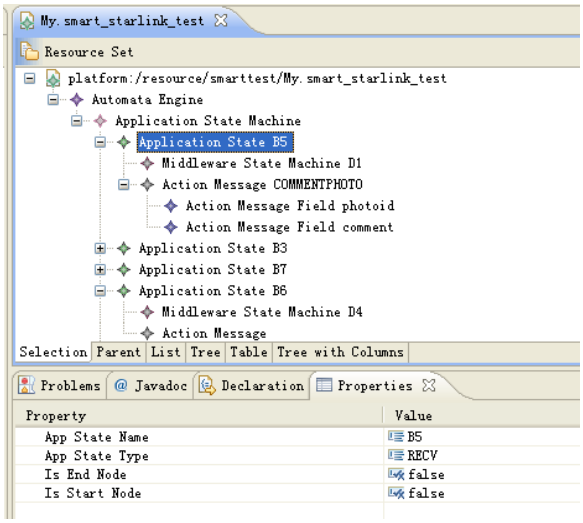


Figure 7 Excerpt of runtime model of photo sharing composite service

B. Case Study

We use the case study of the photo sharing composite service described in Section III, to demonstrate how our approach facilitates runtime management of service composition from the following aspects: (1) providing the comprehensive monitoring functions; (2) simplifying management tasks by model based techniques; (3) applying to the running service on the fly.

Figure 7 shows a fragment of the runtime model of the photo sharing composite service. At the top part, there is a tree view interface for the global scope of the runtime model. Elements can be selected to get and set the values of its properties in the editor at the bottom part. In the picture, we can see that *ApplicationState* "B5" is selected and it has a *MiddlewareStateMachine* and an *ActionMessage* naming COMMENTPHOTO. The COMMENTPHOTO *ActionMessage* class has two *ActionMessageFields*, which indicate the photoid and comment related to this action. If *ActionMessageField* "comment" is selected, we will see in the bottom property view which photo the client-side service is to comment.

Next we will show how to enable user-specific monitoring with QVT model transformation technique. In Figure 7, it can be found that all *ApplicationStates* are listed in the tree view. However, the administrator may only concern about the current state. To satisfy this requirement, we first build a new model named *Monitor*, which has a root class *Monitor* combining a *CurrentState* class. Then we write the following QVT programs in Figure 8(a). The QVT codes specify the relationship between the original PhotoSharing runtime model and the target *Monitor* model. The *CurrentState* class in Monitor model takes the same name with *ApplicationState* in PhotoSharing model on condition that the *ApplicationState* is the current state. As we mentioned, it is also very simple to check the violation of constraints by model techniques. Here we give the example to check the constraint that the length of comment should be no less than 30 characters, as shown in

```
transformation Photosharing2Monitoring(source:PhotoSharing, target:Monitor){
    top relation Application2Current{
        currentState: String;

        checkonly domain source sourceRoot:AutomataEngine{};
        checkonly domain source appState:ApplicationState {
            currentState = appState.name,
            system = sourceRoot
        };

        enforce domain target targetRoot:Monitor{};
        enforce domain target state:CurrentState{
            currentState = state.name,
            system = targetRoot
        };
        when { appState.name = sourceRoot.connectorSM.appCurrentState.name; }
    }
}
```

(a) QVT for customization of monitoring model

```
mapping ActionMessage::check()
    when
    {
        self.has->forAll(m:ActionMessageField |
                         (m.name = "comment" and m.value.length()>30));
    }
```

(b) QVT for checking constraints on comment length

Figure 8 QVT code for monitoring

Figure 8(b).

Finally, we illustrate our runtime model can realize runtime changes of composite service in on-the–fly fashion. Consider a scenario that the IP address of server-side service is changed. This may be due to the reason that in the current network environment, the service residing in the local network can show a better performance. To achieve runtime changes without stopping the service, if the conventional hard-code manner is adopted, the administrator has to traverse every *ApplicationState* related with the server-side service and dig into the *NetworkTransport* class to create a new socket connecting to the new IP. This seems to be an exhausted task. However, with runtime model, administrator just has to write much simpler QVT code similar to that in Figure 8, which changes value of IP address field in all the *NetworkTransport* class with respect to XML-RPC *MiddlewareState*. After synchronization, the next request to the server service will be redirected to the new IP.

VI. Related Works

Most of the literature and practical works related to management of service composition are concerned with Web Service Composition. B. Esfandiari and V. Tosic suggested in [16] that Web Service Composition Management (WSCM) is different from the management of individual Web services (WSM) and as well as Business process management (BPM). They pointed out four requirements for WSCM: service discovery, service selection and contract formation, composition verification, composition management. Our approach meets the last two requirements by model-based verification and management, which implements run-time detection, recovery, and resolution of feature interaction problem and support for adaptation of QoS levels. The other two requirements can be satisfied by integrating our approach with service analysis engines. M. P. Papazoglou et al. summarized the start-of-the-art and research challenges of service composition and service management in [3]. They pointed out that "dynamic and adaptive processes" is one of the most notable challenges for service composition. Techniques should be provided to support self-configuring, -optimizing, -healing, and -adapting. Instead of providing static self adaptive mechanisms, our runtime model enables administrators to adjust the composition's functions at runtime and apply to the system on the fly which makes composite service more adaptive to the changing environment.

Besides Web Service Composition Management, J. Brnsted et al. discussed service composition Issues in pervasive environments[17]. They argued that the goals of pervasive environment service composition include contingency management and device heterogeneity. With the basis of Starlink, our approach can easily address these two issues. Since Starlink covers all the layers related to service execution and is not limited to Web Services, our approach can be used in devices of high heterogeneity. We have explained in the previous sections how the runtime model can handle unexpected errors, which has the same meaning as contingency management.

For the area of model based management of service composition, Fabio Casati et al. developed eFlow system [18] to achieve adaptive and dynamic service composition. eFlow model enables the specification of processes that can automatically configure themselves at run-time according to the nature and type of services available on the Internet. Our approach can realize most of eFlow functionalities but the advantage is that our runtime model is UML compliant which has low learning cost and can be clearly understood by administrators. Another similar work is FARAO, which is a Rule-Based Service Composition[19]. It separates the rule out of BEPL frame, thus increasing the adaptability of the orchestration significantly. This has the same idea as our runtime model, i.e. it is also an external part of the service composition middleware. The use of OCL to restrict the properties of our runtime model is like the rules in FARAO.

VII. Conclusion and Future Work

In this paper, we propose a model-based approach to service composition management at run time. It is based on our previous work on a service composition middleware – Starlink and a runtime system management tool – SM@RT. First, we build the system meta-model of Starlink service composition, specifying what kinds of elements of such compositions can be managed. Second, we build the access model of Starlink service composition, specifying how to manipulate the manageable elements to monitor and modify them. Then with these two models, SM@RT generates the synchronization engine. When the composition is running, the runtime model will reflect the actual state of the composited services. With the help of the runtime model, administrators are able to monitor, check constraints and perform control actions to the composition using any model-based techniques (such as QVT and OCL). Our approach makes the following contributions to service composition management: (1) a more comprehensive view of service composition management; (2) an easy-of-use manner to perform management operations at model level without underlying tedious details; (3) an on-the-fly effect on running system by means of synchronization between model and composite service. We demonstrate by a case study of a photo sharing composite service application that our approach can tackle the challenge of service composition management.

This paper mainly realizes the model-based runtime management of service composition applications. In fact, the synchronization performance cost, and the current interface for administrator requires being more user-friendly. Furthermore, although the photo sharing example has demonstrated the great advantages brought by our approach, it is not strong enough to represent the existing popular services. As future work, we will continue to polish our model and implement our management interface for better usability. At the same time, we also plans to do more testing in performance using real-world complex services.

Acknowledgment

This work was in part funded under the European FP7 ICT FET CONNECT project (http://connect-forever.eu/).

REFERENCES

[1] T. Erl: Service-Oriented Architecture (SOA): Concepts, Technology, and Design. Prentice Hall, 2005.

[2] N. Ibrahim, F. Mouel: A Survey on Service Composition Middleware in Pervasive Environments. International Journal of Computer Science Issues, Volume 1, pp1-12, August 2009.

[3] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann: Service-oriented computing State of the art and research challenges. Computer, Volume: 40 , Issue: 11, pp38- 45, Nov. 2007.

[4] G. Blair, N. Bencomo, R.B. France: Models@ run.time. Computer, Volume: 42 , Issue: 10, pp22- 27, Oct. 2009.

[5] G. Huang, H. Mei, and F.Q. Yang: Runtime recovery and manipulation of software architecture of component-based systems. Auto. Soft. Eng. 13(2) (2006) 257-281.

[6] R. France, B. Rumpe: Model-driven Development of Complex Software: A Research Roadmap. Future of Software Engineering, 2007 (FOSE'07).

[7] A. Occello, A. Dery-Pinna, M. Riveill: A Runtime Model for Monitoring Software Adaptation Safety and its Concretisation as a Service. Models@ runtime, 2008.

[8] H. Song, Y. Xiong, F. Chauvel, G. Huang and Z. Hu, et al. Generating Synchronization Engines between Running Systems and Their Model-Based Views. Models in Software Engineering, Pages 140-154, 2010.

[9] Y. Bromberg, P. Grace, L. Réveillère: Starlink: Runtime Interoperability between Heterogeneous Middleware Protocols. 31st International Conference on Distributed Computing Systems (ICDCS), 2011.

[10] V. Issarny, B. Steffen, B. Jonsson, G. Blair, et. al: Connect challenges: Towards emergent connectors for eternal networked systems. ICECCS, 2009.

[11] Object Management Group. "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)". http://www.omg.org/spec/QVT

[12] Object Management Group. "Object Constraint Language". http://www.omg.org/spec/OCL/2.0/

[13] Object Management Group. The common object request broker: Architecture and specification version 2.0. Technical report, 1995

[14] Apache XML-RPC. http://ws.apache.org/xmlrpc/

[15] F. Budinsky, S. Brodsky, E. Merks: Eclipse Modeling Framework, project address: http://www.eclipse.org/modeling/emf

[16] B. Esfandiari, V. Tosic: Towards a Web Service Composition Management Framework. International Conference on Web Services 2005 (ICWS'05).

[17] J. Brnsted, K.M. Hansen, M. Ingstrup: Service Composition Issues in Pervasive Computing. Pervasive Computing, IEEE, Volume: 9 , Issue: 1, pp62-70, Jan.- March 2010.

[18] F. Casati, S. Ilnicki et al. Adaptive and Dynamic Service Composition in eFlow. Advanced Information Systems Engineering, 2000, Volume 1789/2000, pp13-31.

[19] Weigand, H., Heuvel, W.J.A.M. van den, & Hiel, M: Rule-based service composition and service-oriented business rule management. Proceedings of the International Workshop on Regulations Modelling and Deployment (ReMoD'08) (pp. 1-12). 2008.

[20] H. Song, G. Huang, Y. Xiong, F. Chuavel, Y. Sun, H. Mei. Inferring Meta-Models for Runtime System Data from the Clients of Management APIs. MoDELS 2010.LNCS 6395, 168-182.

[21] H. Song, G. Huang, F. Chauvel, W. Zhang, Y. Sun, W. Shao, H. Mei. Instant and Incremental QVT Transformation for Runtime Models. MODELS2011.

Machine Learning for Emergent Middleware

Amel Bennaceur[1], Valérie Issarny[1], Daniel Sykes[1], Falk Howar[2], Malte Isberner[2], Bernhard Steffen[2], Richard Johansson[3], and Alessandro Moschitti[3]

[1] Inria, Paris-Rocquencourt, France
[2] Technical University of Dortmund, Germany
[3] University of Trento, Italy

Abstract. Highly dynamic and heterogeneous distributed systems are challenging today's middleware technologies. Existing middleware paradigms are unable to deliver on their most central promise, which is offering interoperability. In this paper, we argue for the need to dynamically synthesise distributed system infrastructures according to the current operating environment, thereby generating "Emergent Middleware" to mediate interactions among heterogeneous networked systems that interact in an *ad hoc* way. The paper outlines the overall architecture of Enablers underlying Emergent Middleware, and in particular focuses on the key role of learning in supporting such a process, spanning statistical learning to infer the semantics of networked system functions and automata learning to extract the related behaviours of networked systems.

Keywords: Machine learning, Natural language processing, Automata learning, Interoperability, Automated Mediation

1 Introduction

Interoperability is a fundamental property in distributed systems, referring to the ability for two or more systems, potentially developed by different manufacturers, to work together. Interoperability has always been a challenging problem in distributed systems, and one that has been tackled in the past through a combination of middleware technologies and associated bridging solutions. However, the scope and level of ambition of distributed systems continue to expand and we now see a significant rise in complexity in the services and applications that we seek to support.

Extreme distributed systems challenge the middleware paradigm that needs to face on-the-fly connection of highly heterogeneous systems that have been developed and deployed independently of each other. In previous work, we have introduced the concept of *Emergent Middleware* to tackle the extreme levels of heterogeneity and dynamism foreseen for tomorrow's distributed systems [13, 4].

Emergent Middleware is an approach whereby the necessary middleware to achieve interoperability is not a static entity but rather is generated dynamically as required by the current context. This provides a very different perspective on

middleware engineering and, in particular requires an approach that create and maintain the models of the current networked systems and exploit them to reason about the interaction of these networked systems and synthesise the appropriate artefact, i.e., the emergent middleware, that enable them to interoperate. However, although the specification of system capabilities and behaviours have been acknowledged as fundamental elements of system composition in open networks (especially in the context of the Web [8, 16]), it is rather the exception than the norm to have such rich system descriptions available on the network.

This paper focuses on the pivotal role of learning technologies in supporting Emergent Middleware, including in building the necessary semantic run-time models to support the synthesis process and also in dealing with dynamism by constantly re-evaluating the current environment and context. While learning technologies have been deployed effectively in a range of domains, including in Robotics [27], Natural Language Processing [20], Software Categorisation [26], Model-checking [23], Testing [12], and Interface Synthesis [2], and Web service matchmaking [15], this is the first attempt to apply learning technologies in middleware addressing the core problem of interoperability.

This work is part of a greater effort within the CONNECT project[4] on the synthesis of Emergent Middleware for GMES-based systems that are representative of Systems of Systems. GMES[5] (Global Monitoring for Environment and Security) is the European Programme for the establishment of a European capacity for Earth Observation started in 1998. The services provided by GMES address six main thematic areas: land monitoring, marine environment monitoring, atmosphere monitoring, emergency management, security and climate change. The emergency management service directs efforts towards a wide range of emergency situations; in particular, it covers different catastrophic circumstances: Floods, Forest fires, Landslides, Earthquakes and volcanic eruptions, Humanitarian crises.

For our experiments, we concentrate on joint forest-fire operation that involves different European organisations due to, e.g., the cross-boarder location or criticality of the fire. The target GMES system involves highly heterogeneous NSs, which are connected on the fly as mobile NSs join the scene. Emergent Middleware then need to be synthesised to support such connections when they occur. In the following, we more specifically concentrate on the connection with the Weather Station NS, which may have various concrete instances, ranging from mobile stations to Internet-connected weather service. In addition, Weather Station NSs may be accessed from heterogeneous NSs, including mobile handheld devices of the various people on site and Command and Control —C2— centres (see Figure 1). We show how the learning techniques can serve complementing the base interface description of the NS with appropriate functional and behavioural semantics. It is in particular shown that the process may be fully automated, which is a key requirement of the Emergent Middleware concept.

[4] `http://connect-forever.eu/`
[5] `http://www.gmes.info`

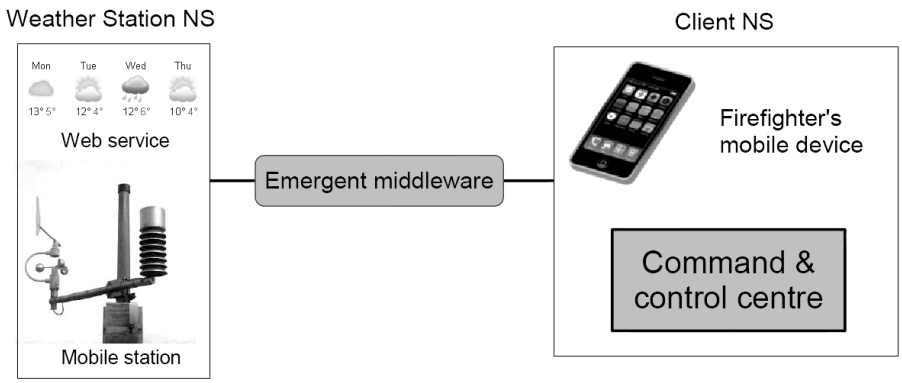


Fig. 1. Heterogeneous Connections with Weather Station NSs

2 Emergent Middleware

Emergent Middleware is synthesised in order to overcome the interoperability issue arising from two independently-developed Networked Systems (NSs). Given two Networked Systems where one implements the functionality required by the other, an Emergent Middleware that mediates application- and middleware-layer protocols implemented by the two NSs is deployed in the networked environment, based on the run-time models of the two NSs and provided that a protocol mediator can indeed be computed. The following section defines the NS model we use to represent the networked systems and reason about their interoperation. Then we present the by *Enablers*, i.e., active software entities that collaborate to realise the Emergent Middleware ensuring their interoperation.

2.1 Networked System Model

The definition of NS models takes inspiration from system models elaborated by the Semantic Web community toward application-layer interoperability. As depicted on Figure 2.(a), the NS model then decomposes into:

– *Interface*: The NS interface provides a microscopic view of the system by specifying fine-grained *actions* (or methods) that can be performed by (i.e., external action required by NS in the environment for proper functioning) and on (i.e., actions provided by the given NS in the networked environment) NS.
There exist many interface definition languages and actually as many languages as middleware solutions. In our approach, we use a SAWSDL-like[6] XML schema. In particular, a major requirement is for interfaces to be annotated with ontology concepts so that the semantics of embedded actions and related parameters can be reasoned about.

[6] `http://www.w3.org/2002/ws/sawsdl/spec/`

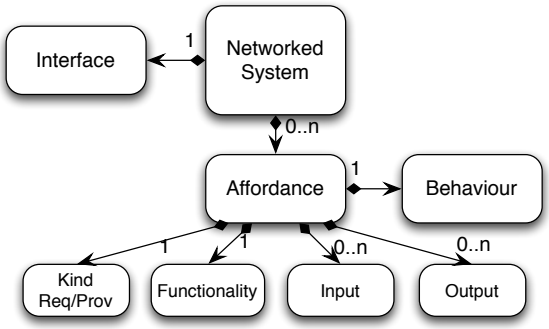


Fig. 2. The Networked System (NS) Model

– *Affordances*: The affordances (*a.k.a. capabilities* in OWL-S [16]) describe the high-level roles an NS plays, e.g., weather station, which are implemented as protocols over the system's observable actions (i.e., actions specified in the NS interface). The specification of an affordance decomposes into:
  - The *ontology-based semantic characterisation* of the high level *Functionality* implemented by the affordance, which is given in terms of the ontology concepts defining the given functionality and of the associated *Input* and *Output*. An affordance is further either *requested* or *provided* by the NS in the networked environment. In the former case, the NS needs to access a remote NS providing the affordance for correct operation; in the latter, the NS may be accessed for the implementation of the given affordance by a remote NS.
  - The affordance's *behaviour* describes how the actions of the interface are co-ordinated to achieve the system's given affordance. Precisely, the affordance behaviour is specified as a process over actions defined in the interface, and is represented as a Labelled Transition System (LTS).

2.2 Emergent Middleware Enablers

In order to produce an Emergent Middleware solution, an architecture of Enablers is required that executes the Emergent Middleware lifecycle. An Enabler is a software component that executes a phase of the Emergent Middleware, co-ordinating with other Enablers during the process.

The Emergent Middleware Enablers are informed by *domain ontologies* that formalise the concepts associated with the application domains (i.e., the vocabulary of the application domains and their relationship) of interest. Three challenging *Enablers* must then be comprehensively elaborated to fully realise Emergent Middleware:

1. The *Discovery Enabler* is in charge of discovering the NSs operating in a given environment. The *Discovery Enabler* receives both the advertisement messages and lookup request messages that are sent within the network

environment by the NSs using legacy discovery protocols (e.g., SLP[7]) thereby allowing the extraction of basic NS models based on the information exposed by NSs, i.e., identification of the NS interface together with middleware used for remote interactions. However, semantic knowledge about the NS must be learned as it is not commonly exposed by NSs directly.

2. The *Learning Enabler* specifically enhances the model of discovered NSs with the necessary functional and behavioural semantic knowledge. The *Learning Enabler* uses advanced learning algorithms to dynamically infer the ontology-based semantics of NSs' affordances and actions, as well as to determine the interaction behaviour of an NS, given the interface description exposed by the NS though some legacy discovery protocol. As detailed in subsequent sections, the Learning Enabler implements both statistical and automata learning to feed NS models with adequate semantic knowledge, i.e., functional and behavioural semantics.

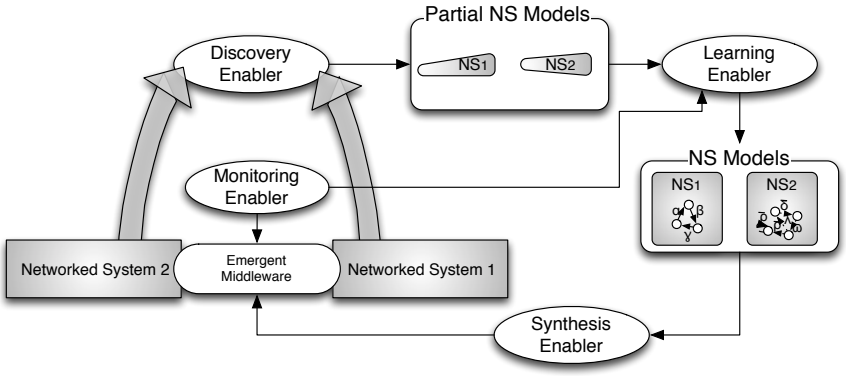


Fig. 3. The Enablers supporting Emergent Middleware

3. The *Synthesis Enabler* dynamically generates the software (i.e., Emergent Middleware) that mediates interactions between two legacy NS protocols to allow them to interoperate. In more detail, once NS models are complete, initial semantic matching of two affordances, that are respectively provided and required by two given NSs, may be performed to determine whether the two NSs are candidates to have an Emergent Middleware generated between them. The semantic matching of affordances is based on the subsumption relationship possibly holding between the concepts defining the functional semantics of the compared affordances.
   Given a functional semantic match of two affordances, the affordances' behaviour may be further analysed to ultimately generate a mediator in case of behavioural mismatch. It is the role of the *Synthesis Enabler* to analyse the behaviour of the two affordances and then synthesise—if applicable—the

[7] `http://www.openslp.org/`

mediator component that is employed by the Emergent Middleware to enable the NSs to coordinate properly to realise the given affordance. For this, the Synthesis Enabler performs automated behavioural matching and mapping of the two models. This uses the ontology-based semantics of actions to say where two sequences of actions in the two behaviours are semantically equivalent; based upon this, the matching and mapping algorithms determine a LTS model that represents the mediator. In few words, for both affordance protocols, the mediator LTS defines the sequences of actions that serve to translate actions from one protocol to the other, further including the possible re-ordering of actions.

The Learning phase is a continuous process where the knowledge about NSs is enriched over time, thereby implying that Emergent Middleware possibly needs to adapt as the knowledge evolves. In particular, the synthesised Emergent Middleware is equipped with monitoring probes that gather information on actual interaction between connected systems. This observed *Monitoring Data* is delivered to the Learning Enabler, where the learned hypotheses about the NSs' behaviour are compared to the observed interactions. Whenever an observation is made by the monitoring probes that is not contained in the learned behavioural models, another iteration of learning is triggered, yielding refined behavioural models. These models are then used to synthesise and deploy an evolved Emergent Middleware.

3 Machine Learning: A Brief Taxonomy

Machine learning is the discipline that studies methods for automatically inducing functions (or system of functions) from data. This broad definition of course covers an endless variety of subproblems, ranging from the least-squares linear regression methods typically taught at undergraduate level [21] to advanced structured output methods that learn to associate complex objects in the input [18] with objects in the output [14] or methods that infer whole computational structures [10]. To better understand the broad range of machine learning, one must first understand the conceptual differences between learning setups in terms of their prerequisites:

- *Supervised learning* is the most archetypical problem setting in machine learning. In this setting, the learning mechanism is provided with a (typically finite) set of labelled examples: a set of pairs $T = \{(x, y)\}$. The goal is to make use of the example set T to induce a function f, such that $f(x) = y$, for future unseen instances of (x, y) pairs (see for example [21]). A major hurdle in applying supervised learning is the often enormous effort of labelling the examples.
- *Unsupervised learning* lowers the entry hurdle for application by requiring only unlabelled example sets, i.e., $T = \{x\}$. In order to be able to come up with anything useful when no supervision is provided, the learning mechanism needs a bias that guides the learning process. The most well-known

example of unsupervised learning is probably k-means clustering, where the learner learns to categorise objects into broad categories even though the categories were not given a priori. Obviously, the results of unsupervised learning cannot compete with those of supervised learning.
- *Semi-supervised learning* is a pragmatic compromise. It allows one to use a combination of a small labelled example set $T_s = \{(x, y)\}$ together with a larger unlabelled example set $T_u = \{x\}$ in order to improve on both the plain supervised learner making use of T_s only and the unsupervised learner using all available examples.
- *Active learning* puts the supervisor in a feedback loop: whenever the (active) learner detects a situation where the available test set is inconclusive, the learner actively constructs complementing examples and asks the supervisor for the corresponding labelling. This learning discipline allows a much more targeted learning process, since the active learner can focus on the important/difficult cases (see for example [5]). The more structured the intended learning output is, the more successful active learning will be, as the required structural constraints are a good guide for the active construction of examples [3]. It has been successfully used in practice for inferring computational models via testing [11, 10].

Learning technology has applicability in many domains. The next sections concentrate on the learning-based techniques that we are developing to enable the automated inference of semantic knowledge about Networked Systems, both functional and behavioural. The former relies on *statistical learning* while the latter is based on *automata learning*.

4 Statistical Learning for Inferring NS Functional Semantics

As discussed in Section 2.2, the first step in deciding whether two NSs will be able to interoperate consists in checking the compatibility of their *affordances* based on their functional semantics (i.e., ontology concepts characterising the purpose of the affordance). Then, in the successful cases, behavioural matching is performed so as to synthesise required mediator. This process highlights the central role of the functional matching of affordances in reducing the overall computation by acting as a kind of filter for the subsequent behavioural matching. Unfortunately, legacy applications do not normally provide affordance descriptions. We must therefore rely upon an engineer to provide them manually, or find some automated means to extract the probable affordance from the interface description. Note that it is not strictly necessary to have an absolutely correct affordance since falsely-identified matches will be caught in the subsequent detailed checks.

Since the interface is typically described by textual documentation, e.g., XML documents, we can capitalise on the long tradition of research in *text categorisation*. This studies approaches for automatically enriching text documents with

semantic information. The latter is typically expressed by topic categories: thus text categorisation proposes methods to assign documents (in our case, interface descriptions) to one or more categories. The main tool for implementing modern systems for automatic document classification is machine learning based on vector space document representations.

In order to be able to apply standard machine learning methods for building categorizers, we need to represent the objects we want to classify by extracting informative *features*. Such features are used as indications that an object belongs to a certain category. For categorisation of documents, the standard representation of features maps every document into a vector space using the *bag-of-words* approach [25]. In this method, every word in the vocabulary is associated with a dimension of the vector space, allowing the document to be mapped into the vector space simply by computing the occurrence frequencies of each word. For example, a document consisting of the string "get Weather, get Station" could be represented as the vector $(2, 1, 1, \ldots)$ where, e.g., 2 in the first dimension is the frequency of the "get" token. The bag-of-words representation is considered the standard representation underlying most document classification approaches. In contrast, attempts to incorporate more complex structural information have mostly been unsuccessful for the task of categorisation of single documents [22] although they have been successful for complex relational classification tasks [19].

However, the task of classifying interface descriptions is different from classifying raw textual documents. Indeed, the interface descriptions are *semi-structured* rather than unstructured, and the representation method clearly needs to take this fact into account, for instance, by separating the vector space representation into regions for the respective parts of the interface description. In addition to the text, various semi-structured identifiers should be included in the feature representation, e.g., the names of the method and input parameters defined by the interface. The inclusion of identifiers is important since: (*i*) the textual content of the identifiers is often highly informative of the functionality provided by the respective methods; and (*ii*) the free text documentation is not mandatory and may not always be present.

For example, if the functionality of the interface are described by an XML file written in WSDL, we would have tags and structures, as illustrated by the text fragment below, which relates to a NS implementing a weather station and is part of the GMES scenario detailed in the next section on experiments:

```
<wsdl:message name="GetWeatherByZipCodeSoapIn">
    <wsdl:part name="parameters"
        element="tns:GetWeatherByZipCode" />
</wsdl:message>
 <wsdl:message name="GetWeatherByZipCodeSoapOut">
   <wsdl:part name="parameters"
       element="tns:GetWeatherByZipCodeResponse"/>
</wsdl:message>
```

It is clear that splitting the CamelCase identifier `GetWeatherStation` into the tokens `get`, `weather`, and `station`, would provide more meaningful and

generalised concepts, which the learning algorithm can use as features. Indeed, to extract useful word tokens from the identifiers, we split them into pieces based on the presence of underscores or CamelCase; all tokens are then normalised to lowercase.

Once the feature representation is available, we use it to learn several classifiers, each of them specialised to recognise if the WSDL expresses some target semantic properties. The latter can also be concepts of an ontology. Consequently, our algorithm may be used to learn classifiers that automatically assign ontology concepts to actions defined in NS interfaces. Of course, the additional use of domain (but at the same time general) ontologies facilitates the learning process by providing effective features for the interface representation. In other words, WSDL, domain ontologies and any other information contribute to defining the vector representation used for training the concept classifiers.

5 Automata Learning For Inferring NS Behavioural Semantics

Automata learning can be considered as a key technology for dealing with *black box* systems, i.e., systems that can be observed, but for which no or little knowledge about the internal structure or even their intent is available. Active Learning (*a.k.a* regular extrapolation) attempts to construct a deterministic finite automaton that matches the behaviour of a given target system on the basis of test-based interaction with the system. The popular L^* algorithm infers Deterministic Finite Automata (DFAs) by means of *membership queries* that test whether certain strings (potential runs) are contained in the target system's language (its set of runs), and *equivalence queries* that compare intermediately constructed hypothesis automata for language equivalence with the target system.

In its basic form, L^* starts with a hypothesis automaton that treats all sequences of considered input actions alike, i.e., it has one single state, and refines this automaton on the basis of query results, iterating two main steps: (1) refining intermediate hypothesis automata using membership queries until a certain level of "consistency" is achieved (*test-based modelling*), and (2) testing hypothesis automata for equivalence with the target system via equivalence queries (*model-based testing*). This procedure successively produces state-minimal deterministic (hypothesis) automata consistent with all the encountered query results [3]. This basic pattern has been extended beyond the domain of learning DFAs to classes of automata better suited for modelling reactive systems in practice. On the basis of active learning algorithms for Mealy machines, inference algorithms for I/O-automata [1], timed automata [7], Petri Nets [6], and Register Automata [10], i.e., restricted flow graphs, have been developed.

While usually models produced by active learning are used in model-based verification or some other domain that requires complete models of the system under test (e.g., to prove absence of faults), here the inferred models serve as a basis for the interaction with the system for Emergent Middleware synthesis.

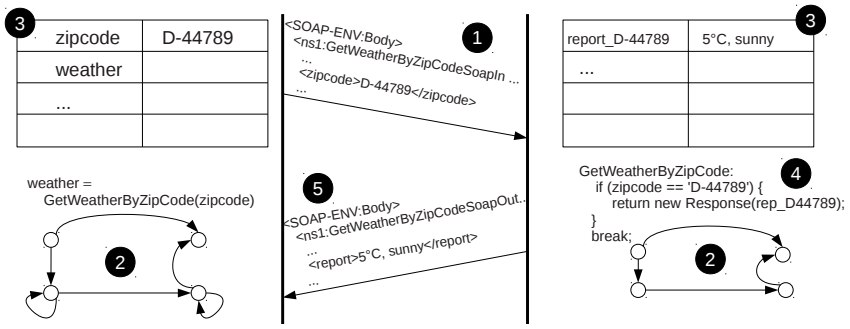


Fig. 4. Communicating Components

This special focus poses unique requirements on the inferred models (discussed in detail in [9]), which become apparent in the following prototypical example.

Figure 4 shows a typical interoperability scenario where two NSs are actual implementations of their specified interfaces. The NS on the right implements a weather service that provides weather forecasts for regions identified by ZIP codes. The NS on the left is a matching client. The two NSs communicate via SOAP protocol messages (1), (5), and together realise some protocol, which comprises a control part (2), and a data part (3) at both NSes. The data parts may be best described as a set of local variables or registers. The control part can be modelled as a labeled transition system with actual blocks of code labelling the transitions (4). Each code block of Fig. 4 would consist of an entry point for one interface method (e.g., `GetWeatherByZipCode`), conditions over parameters and local variables (e.g., comparing ZIP codes), assignments and operations on local variables (e.g., storing returned weather data), and a return statement.

To infer the behaviour of one NS (say, the right one from Fig. 4), the role of the other NS has to be undertaken by a learning algorithm, which is aware of the interface alphabet of the NS whose affordance's behaviour is to be learned. This interface alphabet is derived automatically from the interface description of the NS under scrutiny. A test-driver is then instantiated by the Learning Enabler, translating the alphabet symbols to remote invocations of the NS to be learned.

Now, to capture the interaction of the two NSs faithfully, two phenomena have to be made explicit in the inferred models:

– *Preconditions of Primitives:* Usually real systems operate on communication primitives that contain data values relevant to the communication context and have a direct impact on the exposed behaviour. Consider as an example session identifiers or sequence numbers that are negotiated between the communication participants and included in every message. The models have to make explicit causal relations between data parameters that are used in the communication (e.g, the exact session identifier that is returned when opening a new session has to be used in subsequent calls).

– *Effects of Primitives:* The learned models will only be useful for Emergent Middleware (mediator) synthesis within a given semantic context. Most NSs have well-defined purposes as characterised by affordances (e.g., getting localised weather information). A subset of the offered communication primitives, when certain preconditions are met, will lead to successful conclusion of this purpose. This usually will not be deducible from the communication with a system: an automata learning algorithm in general cannot tell error messages and regular messages (e.g., weather information) apart. In such cases, information about effects of primitives rather has to be provided as an additional (semantic) input to the learning algorithm (e.g., in terms of ontologies [4]), as supported by the semantically annotated interface descriptions of NSes.

Summarizing, in the context of Emergent Middleware, especially dealing with parameters and value domains, and providing semantic information on the effect of communication primitives, are aspects that have to be addressed with care. We have reaffirmed this analysis in a series of experiments on actual implementations of NSs.

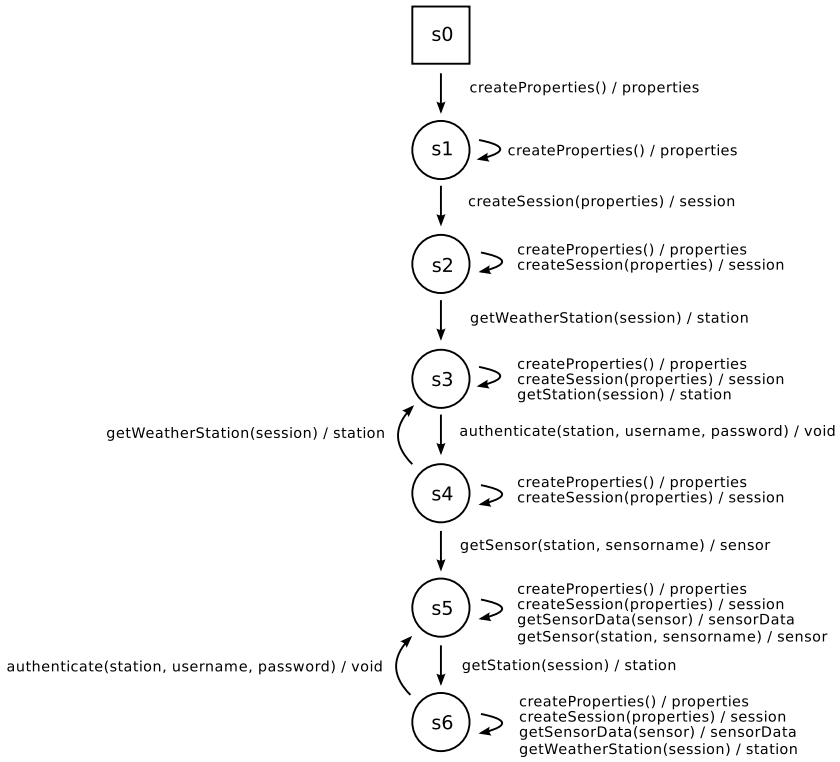


Fig. 5. Behavioural Model of the Weather Station Sensor Network Service – Starting State is `s0`

The automata learning technique is provided by LearnLib [17, 24], a component-based framework for automata learning. In the produced model, each transition consists of two parts, separated by a forward-slash symbol: on the left hand side an abstract parameterised symbol is denoted, while on the right hand side the named variable storing the invocation result is specified. Figure 5 depicts the behavioural description of the weather station, which was learned in 31 seconds on a portable computer, using 258 MQs.

The model correctly reflects the steps necessary, e.g., to read sensor data: `createProperties`, `createSession`, `getWeatherStation`, `authenticate` and `getSensor` have to be invoked before `getSensorData` can be called successfully. Additionally, the actual realisation of authentication, which cannot be deduced from the interface specification alone, is revealed in the inferred model. When simply looking at the parameter types, the action `getSensor` should be invocable directly after the `getWeatherStation` primitive. However, in reality `getSensor` is guarded by an authentication mechanism, meaning that `authenticate` has to be successfully invoked beforehand. Also, from the model, it is easily deducible that the `authenticate` action will indeed merely affect the provided station data object (and not, e.g., the whole session): requesting a new station data object will always necessitate another authentication step before `getSensor` can be invoked again, as that action requires an authenticated station data object.

6 Conclusion

This paper has presented the central role of learning in supporting the concept of Emergent Middleware, which revisits the middleware paradigm to sustain interoperability in increasingly heterogeneous and dynamic complex distributed systems. The production of Emergent Middleware raises numerous challenges, among which dealing with the *a priori* minimal knowledge about networked systems that is available to the generation process. Indeed, semantic knowledge about the interaction protocols run by the Networked Systems is needed to be able to reason and compose protocols in a way that enable NSs to collaborate properly. While making such knowledge available is increasingly common in Internet-worked environments (e.g., see effort in the Web service domain), it remains absent from the vast majority of descriptions exposed for the Networked Systems that are made available over the Internet. This paper has specifically outlined how powerful learning techniques that are being developed by the scientific community can be successfully applied to the Emergent Middleware context, thereby enabling the automated learning of both functional and behavioural semantics of NSs. In more detail, this paper has detailed how statistical and automata learning can be exploited to enable on-the-fly inference of functional and behavioural semantics of NSs, respectively.

Our experiments so far show great promise with respect to the effectiveness and efficiency of machine learning techniques applied to realistic distributed systems such as in the GMES case. Our short-term future work focuses on the fine tuning of machine learning algorithms according to the specifics of the networked

systems as well as enhancing the learnt models with data representations and non-functional properties, which can result in considerable gains in terms of accuracy and performance. In the mid-term, we will work on the realisation of a continuous feedback loop from real-execution observations of the networked systems to update the learnt models dynamically as new knowledge becomes available and to improve the synthesised emergent middleware accordingly.

Acknowledgments. This research has been supported by the EU FP7 projects: CONNECT – Emergent Connectors for Eternal Software Intensive Networking Systems (project number FP7 231167), EternalS – "Trustworthy Eternal Systems via Evolving Software, Data and Knowledge" (project number FP7 247758) and by the EC Project, LIMOSINE – Linguistically Motivated Semantic aggregation engiNes (project number FP7 288024).

References

1. Aarts, F., Vaandrager, F.: Learning I/O Automata. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory, Lecture Notes in Computer Science, vol. 6269, pp. 71–85. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-15375-4_6
2. Alur, R., Cerny, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for Java classes. In: Proc. POPL '05 (2005)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
4. Blair, G., Bennaceur, A., Georgantas, N., Grace, P., Issarny, V., Nundloll, V., Paolucci, M.: The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems. In: Middleware 2011 - 12th International Middleware Conference (2011)
5. Cohn, D.A., Ghahramani, Z., Jordan, M.I.: Active learning with statistical models. J. Artif. Intell. Res. (JAIR) 4, 129–145 (1996)
6. Esparza, J., Leucker, M., Schlund, M.: Learning workflow petri nets. vol. 113, pp. 205–228 (2011)
7. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of Event-Recording Automata Using Timed Decision Trees. In: Proc. CONCUR 2006, 17^{th} Int. Conf. on Concurrency Theory. pp. 435–449 (2006)
8. Heß, A., Kushmerick, N.: Learning to attach semantic metadata to web services. In: International Semantic Web Conference. pp. 258–273 (2003)
9. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On handling data in automata learning - considerations from the connect perspective. In: ISoLA (2). pp. 221–235 (2010)
10. Howar, F., Steffen, B., Jonsson, B., Cassel, S.: Inferring canonical register automata. In: VMCAI. pp. 251–266 (2012)
11. Howar, F., Steffen, B., Merten, M.: Automata learning with automated alphabet abstraction refinement. In: VMCAI. pp. 263–277 (2011)
12. Hungar, H., Margaria, T., Steffen, B.: Test-based model generation for legacy systems. In: Test Conference, 2003. Proceedings. ITC 2003. International. vol. 1, pp. 971–980 (30-Oct 2, 2003)

13. Issarny, V., Steffen, B., Jonsson, B., Blair, G., Grace, P., Kwiatkowska, M., Calinescu, R., Inverardi, P., Tivoli, M., Bertolino, A., Sabetta, A.: CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In: 14th IEEE International Conference on Engineering of Complex Computer Systems (2009)
14. Joachims, T., Hofmann, T., Yue, Y., Yu, C.N.J.: Predicting structured objects with support vector machines. Commun. ACM 52(11), 97–104 (2009)
15. Katakis, I., Meditskos, G., Tsoumakas, G., Bassiliades, N., Vlahavas, I.P.: On the combination of textual and semantic descriptions for automated semantic web service classification. In: AIAI. pp. 95–104 (2009)
16. Martin, D.L., Burstein, M.H., McDermott, D.V., McIlraith, S.A., Paolucci, M., Sycara, K.P., McGuinness, D.L., Sirin, E., Srinivasan, N.: Bringing semantics to web services with OWL-S. In: World Wide Web. pp. 243–277 (2007)
17. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next generation learnlib. In: TACAS. pp. 220–223 (2011)
18. Moschitti, A.: Efficient convolution kernels for dependency and constituent syntactic trees. In: ECML. pp. 318–329 (2006)
19. Moschitti, A.: Kernel methods, syntax and semantics for relational text categorization. In: Proceedings of ACM 17th Conference on Information and Knowledge Management (CIKM). Napa Valley, United States (2008)
20. Moschitti, A.: Kernel-based machines for abstract and easy modeling of automatic learning. In: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-11 (2011)
21. Moschitti, A.: Kernel-based machines for abstract and easy modeling of automatic learning. In: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-11. pp. 458–503 (2011)
22. Moschitti, A., Basili, R.: Complex linguistic features for text classification: A comprehensive study. In: Proceedings of the 26th European Conference on Information Retrieval Research (ECIR 2004). pp. 181–196. Sunderland, United Kingdom (2004)
23. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: FORTE. pp. 225–240 (1999)
24. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. Int. J. Softw. Tools Technol. Transf. 11(5), 393–407 (2009)
25. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Tech. Rep. TR74-218, Department of Computer Science, Cornell University, Ithaca, New York (1974)
26. Selby, R., Porter, A.: Learning from examples: generation and evaluation of decision trees for software resource analysis. Software Engineering, IEEE Transactions on 14(12) (1988)
27. Stone, P., Veloso, M.: Multiagent systems: A survey from a machine learning perspective. Autonomous Robots 8 (2000)

Bibliography

[1] Part-whole relations in owl ontologies. http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/.

[2] A. Bennaceur, G. S. Blair, F. Chauvel, N. Georgantas, P. Grace, V. Nundloll, M. Paolucci, R. Saadi, and D. Sykes. Intermediate CONNECT architecture. Technical Report D1.2, CONNECT ICT FET IP Project, February 2011.

[3] A. Bennaceur, G. S. Blair, F. Chauvel, N. Georgantas, P. Grace, V. Nundloll, M. Paolucci, R. Saadi, and D. Sykes. Revised CONNECT architecture. Technical Report D1.3, CONNECT ICT FET IP Project, February 2012.

[4] A. Bennaceur, J. Richard, M. Alessandro, S. Romina, D. Sykes, R. Saadi, and V. Issarny. Inferring Affordances Using Learning Techniques. In *International Workshop on Eternal Systems (EternalS'11)*, Budapest, Hongrie, 2011.

[5] A. Bertolino, G. Blair, F. Chauvel, C. F. Cortes, N. Georgantas, P. Grace, F. Howar, T. Huyn, B. Jonsson, M. Paolucci, A. Pathak, B. Souville, and M. Tivoli. Initial CONNECT architecture. Technical Report D1.1, CONNECT ICT FET IP Project, February 2010.

[6] G. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In F. Kon and A.-M. Kermarrec, editors, *Middleware 2011*, volume 7049 of *Lecture Notes in Computer Science*, pages 410–430. Springer Berlin / Heidelberg, 2011.

[7] G. S. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, and M. Paolucci. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In F. Kon and A.-M. Kermarrec, editors, *Middleware*, volume 7049 of *Lecture Notes in Computer Science*, pages 410–430. Springer, 2011.

[8] A. Borgida. Towards the systematic development of description logic reasoners: Clasp reconstructed. In *KR*, pages 259–269, 1992.

[9] D. Bromberg, P. Grace, L. Reveillere, and G. Blair. Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In *The 12th International Middleware Conference*, 2011.

[10] Y.-D. Bromberg, P. Grace, and L. Réveillère. Starlink: runtime interoperability between heterogeneous middleware protocols. In *Proceedings of 31th International Conference on Distributed Computing Systems, ICDCS (IEEE)*., pages 446 –455, June 2011.

[11] CONNECT Consortium. Emergent connector for eternal software intensive networked systems. Fet proactive 6: Ict forever yours description of work, CONNECT, 2008.

[12] CONNECT Consortium. Consolodated dependability framework. Technical Report D5.3, CONNECT, February 2011.

[13] CONNECT Consortium. Dynamic connector synthesis - revised prototype implementation. Technical Report D3.3, CONNECT, February 2011.

[14] CONNECT Consortium. Assessment report. Technical Report D6.2, CONNECT, February 2012.

[15] CONNECT Consortium. Finalised dependability framework and evaluation results. Technical Report D5.4, CONNECT, November 2012.

[16] CONNECT Consortium. Integration in the CONNECT architecture. Technical Report D4.4, CONNECT, February 2012.

[17] N. Drummond, A. L. Rector, R. Stevens, G. Moulton, M. Horridge, H. Wang, and J. Seidenberg. Putting OWL in order: Patterns for sequences in OWL. In *OWLED*, 2006.

[18] A. B. Emil Andriescu, P. I. (and Valérie Issarny, R. Spalazzese, and R. Speicys-Cardoso. Dynamic connector synthesis: Principles, methods, tools and assessment. Technical Report D3.4, CONNECT ICT FET IP Project, February 2012.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[20] N. Georgantas, S. Hachem, V. Issarny, M. Autili, D. Di Ruscio, P. Inverardi, M. Tivoli, D. Athanasopoulos, P. Vasilliadis, A. Zarras, D. Batista, and S. Carlos Eduardo Moreira Dos. Initial Architectural Style for CHOReOS Choreographies (D1.3), Oct. 2011.

[21] P. Grace, Y.-D. Bromberg, L. Réveillère, and G. S. Blair. Overstar: An open approach to end-to-end middleware services in systems of systems. In P. Narasimhan and P. Triantafillou, editors, *Middleware*, volume 7662 of *Lecture Notes in Computer Science*, pages 229–248. Springer, 2012.

[22] T. Gu, H. Pung, et al. A middleware for building context-aware mobile services. In *Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th*, volume 5, pages 2656–2660. Ieee, 2004.

[23] M. Hepp. Possible ontologies: How reality constrains the development of relevant ontologies. *Internet Computing, IEEE*, 11(1):90–96, 2007.

[24] A. Heß and N. Kushmerick. Learning to attach semantic metadata to web services. In *ISWC*, pages 258–273, 2003.

[25] H. Hirsh and D. Kudenko. Representing sequences in description logics. In *AAAI/IAAI*, pages 384–389, 1997.

[26] G. Huang, H. Song, and H. Mei. Sm@rt: Applying architecture-based runtime management of internetware systems. *International Journal of Software and Informatics*, 3(4):439–464, 2009.

[27] J. Hunter. Enhancing the semantic interoperability of multimedia through a core ontology. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(1):49–58, 2003.

[28] T. Huynh and A. L. (eds.). Experiment scenarios, prototypes and report iteration 2. Technical Report D6.3, CONNECT ICT FET IP Project, February 2012.

[29] M. Klusch, P. Kapahnke, and I. Zinnikus. SAWSDL-MX2: A machine-learning approach for integrating semantic web service matchmaking variants. In *ICWS*, pages 335–342, 2009.

[30] N. Oldham, C. Thomas, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework with machine learning classification. In *SWSWPC*, 2004.

[31] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723, 2011.

[32] H. Song, G. Huang, F. Chauvel, W. Zhang, Y. Sun, W. Shao, and H. Mei. Instant and incremental qvt transformation for runtime models. In *MODELS*, pages 273–288, 2011.

[33] H. Song, G. Huang, Y. Xiong, F. Chuavel, Y. Sun, and H. Mei. Inferring meta-models for runtime system data from the clients of management apis. In *MODELS*, volume LNCS 6395, pages 168–182, 2010.

[34] A. Varzi. Parts, wholes, and part-whole relations: The prospects of mereotopology. *Data & Knowledge Engineering*, 20(3):259–286, 1996.

[35] Y. Wu, G. Huang, H. Song, and Y. Zhang. Model driven configuration of fault tolerance solutions for component-based software system. In *MODELS*, pages 514–530, 2012.